

Programming of Planning as Modelling in AI

By Paul Almond, 28 December 2006

Website: www.paul-almond.com
Email: info@paul-almond.com

© Copyright Paul Almond, 2006. All Rights Reserved.

Programming of Planning As Modelling in AI

By Paul Almond, 28 December 2006.

Introduction

In previous articles I suggested the *planning as modelling* approach to planning in artificial intelligence (AI). This provides planning in an AI system by using the AI's modelling system to produce probabilistic *predictions* of future behaviour that are equivalent to *planning* of future behaviour. Planning as Modelling was discussed in *Planning as Modelling in AI* [1], *How AI Would Work* [2], *Occam's Razor Part 9: Representation and Planning of Actions in Artificial Intelligence* [3] and *AI as a Boundary System* [4].

This article will provide example computer programs in Pascal for planning as modelling.

The Approach in this Article

The programs provided here cannot be used in their current form because software is not provided to implement the modelling system. Planning as modelling does not specify how to make a modelling system, but how a modelling system making probabilistic predictions can be used to plan the actions of an AI system, eliminating the need for a separate planning system. The computer code in this article therefore assumes that a working, probabilistic model is available. Although the implementation of the modelling system is omitted, subroutines are given to act as an interface between the modelling system and the program using it. What is provided is what would be a basic, working AI system *if* the probabilistic modelling system were added.

Some readers will probably say that is easy to claim a complete AI system design if you do not have to give all the details, but planning as modelling is not claimed to be a complete AI system: it is merely a way of getting a modelling system to do the planning as well. In other articles [2,3,5,6,7,8] I have given more consideration to how the modelling system might work, but planning as modelling does not depend on this.

There are two example programs – one in Standard Pascal and one in Object Pascal/The Delphi Programming Language (the version of Pascal used by Borland's Delphi Pascal compiler) – in the appendices of this article. The main text of this article will describe how these programs work. Readers who prefer just to see how something works, particularly those who have read *Planning as Modelling in AI* [1], may wish to go to the appendices first. The programs contain some sections that are commented “{The code here is specific to the modelling system.}” where code would need to be placed to add the functionality of a specific modelling system and provide a working AI system. These programs might be considered a proposed framework for an AI system.

The example programs in this article assume that each input or output of the system is a single binary digit (bit). I have generally discussed planning as modelling for such systems, but this does not have to be the case. The software could easily be modified to have any number of branches from each search tree node – one for each possible input or output value. The example programs

also use a tree search in the output vectoring system. Planning as modelling does not require a tree search, but it seems a sensible way of doing it and I generally imagine it like this.

In earlier articles about planning as modelling [2,3], the search performed by the output vectoring system was described as a search of the set of possible future sequences of outputs: inputs were omitted. In the last article *Planning as Modelling in AI* [1] I corrected this and described the output vectoring system as searching the set of possible future sequences of inputs and outputs. This correction continues to apply in this article.

How Planning as Modelling Works

Planning as modelling uses an AI system's modelling system to perform both modelling and planning functions without the need for a separate planning system.

The modelling system observes past inputs and outputs of the AI system and makes probabilistic predictions of future inputs *and outputs*. *The AI system is predicting what will happen in reality and its own behaviour*: planning as modelling makes no distinction between them as the system's observations of its own outputs are treated as just more inputs about which predictions can be made.

When an output is required a separate system, the *output vectoring system*, performs a search to determine the optimum output. The search is of the set of possible future sequences of inputs and outputs that the AI system can make and receive. This search can be made using a tree search – as is done with chess algorithms [11]. In such a tree search each branch would correspond to a particular input or output being received or made at some instant of time and the search tree would explore possible future inputs and outputs in chronological order.

The search does not give equal priority to all possible sequences of future inputs and outputs. Instead, *the search is constrained by the modelling system's predictions*. More likely futures are explored in greater detail than less likely ones. The likelihood of a particular future is obtained by multiplying together all the probabilities of the individual inputs and outputs occurring as they do in that future and the modelling system provides these probability values.

If a tree search is used then the constraint from the model can be applied by using a *cut-off probability*. The tree search computes a *running probability* for each node. At the start of the search tree this has a value of 1, and each time the search tree goes down a new branch for a particular input or output having a particular value then the running probability is multiplied by the probability of the input or output occurring with that value (provided by the modelling system) to get the running probability for the new node. The running probability therefore represents the probability that a sequence of inputs and outputs will actually occur to produce the situation described by a particular node of the search tree. Paths through the search tree are prioritised according to the running probability: a path through the search tree is never examined if one with a lower running probability is discarded. New branches of the search tree are only extended provided that the running probability for this node does not exceed the cut-off probability. Terminal nodes of the search tree are evaluated by a situational evaluation function which gives a score indicating the desirability of the situation described by the model. This score is “backed up” the search tree.

Planning as modelling also uses *prioritization control outputs*. These are special pseudo-outputs generated by the modelling system, observed by it and predicted by it in the same way as conventional outputs. The only difference is that instead of being sent to the outside world as conventional outputs they are sent into the modelling system as instructions to control its prioritization of computing resources to deal with the *carpet texture problem*.

Why Planning as Modelling Works

It may seem that this is just the uninteresting suggestion of adding a planning system onto a modelling system. There are two important aspects of this, however:

- The modelling system makes probabilistic predictions of the system's inputs *and outputs*.
- These predictions constrain the search involved in "planning".

The constraint provided by the modelling system dictates how the tree search proceeds. This means that the modelling system is dictating the results that the tree search will produce.

The modelling system is doing the planning.

The modelling system's *predictions* of the AI system's future behaviour are being used to *direct* its future behaviour. This can be summarized, briefly, as follows:

Planning is prediction.

Purpose of the Output Vectoring System

If planning occurs in the modelling system then the output vectoring system, despite superficial resemblance to a "planning system" is not really the AI system's "planning system" at all. This raises the issue of what the output vectoring system's purpose is. It has two purposes:

- to slightly improve the behaviour "planned" by the modelling system's predictions
- to provide stability for the system at a later stage

The predictions for the AI system's future inputs and output are based on the AI system's previous inputs and outputs. If the system's previous behaviour is desirable this means that future behaviour should also be desirable, but there is no reason why the behaviour should initially be desirable. The output vectoring system deals with this by trying to find optimum behaviour, within the constraint of the modelling system's predictions. The output vectoring system's search can improve on previous behaviour because it not only has the previous behaviour as a guide, but is performing a search as well. If it manages to find even slightly better behaviour then the system's own observations of making the outputs associated with this improved behaviour will form part of the history of inputs and outputs which will be used to constrain future searches by the output vectoring system.

An important aspect of this process is that the improvement itself will form a pattern of previous behaviour on which future behaviour will be modelled.

One way of thinking about this, and it is a little strange, is to regard the system as generating its future behaviour by modelling itself and the output vectoring system as cheating by finding ways of improving this behaviour so that when the system is modelling its future behaviour from previous behaviour it gets tricked into thinking it was smarter than it really was when it produced the previous behaviour.

Interaction with the Modelling System

Interaction as Services

The first articles on planning as modelling [2,3,4] described it in the context of a hierarchical AI system. In the last article [1] I gave a more general view of planning as modelling, independent of the context of any particular modelling system. This will continue in this article, although this does not mean that the modelling system should not be hierarchical – merely that I want to provide a general solution for planning.

In the last article the interaction between the output vectoring system and the model was described in terms of the input and output history and prediction collection or combined probability collection. These can be considered as data structures containing sets of previous and future inputs and outputs into which we put observations of known inputs and outputs for the modelling system to observe and from which we extract its probabilistic predictions.

These data structures are a hangover from when planning as modelling was being described solely in the context of a hierarchical system. They are what is left of the bottom level of this hierarchy, but instead of saying that they must be the bottom level of a hierarchical system we say that they are used as the interface with a modelling system that works in an unspecified way.

In this article I will introduce an improvement in the way that interaction with the modelling system is described. Rather than use data structures containing entire sets of probabilities we need only deal with a single input or output event at a time and the modelling system merely needs to provide services to this effect.

We only need to deal with input and output events singly because they are dealt with in chronological order:

- When the modelling system is being updated with the results of real input or output events (when the modelling system is told what the values of real inputs or outputs are) these updates occur in chronological order as the real input or output events happen.
- When the modelling system is being updated with the results of hypothetical real input or output events (when the modelling system is told what the values of hypothetical inputs or outputs are) these updates will usually occur in chronological order as they do, for example, if a tree search approach is used: each branching of the search tree corresponds to the next (in a temporal sense) occurrence of an input or output.

The modelling system does not need to store the details of all the input events and output events that it has been “told about”: It could store some abstraction of many input and output and output

events internally, or it may determine that a particular input event or output event is irrelevant and can be disregarded completely. None of this matters outside the modelling system. Once the modelling system has been informed that a particular input event or output event has occurred with some value then it can be assumed that the modelling system will, in its future predictions, take account of this input or output event, or take account of some abstraction of this event and others, or ignore it completely if it is irrelevant: all of this is specific to the modelling system.

This lack of any need to do more than process inputs and outputs in chronological order frees us from some complexity in the interaction of the modelling system with the rest of the AI system. All we have to bother about is the input or output that is happening *now* – whether this is a real “now” for the system’s real inputs and outputs or a hypothetical “now” reached at some point in the output vectoring system’s search. The modelling system needs a way of telling us about the probabilities associated with this input or output event. We also need some way of moving from one event to the next chronologically and a way of making hypothetical inputs and outputs in the output vectoring system’s search. This can be provided by a simple set of services, all of which are based on the idea of the *current event*.

The Current Event

Input events and output events occur in chronological order. At any time the modelling system is considering a particular input or output event as being the *current event*. The current event is not necessarily the input or output event that is happening now in the real world. The current event is the input or output event for which the following apply:

- The modelling system needs to make probabilistic predictions. If the modelling system is instructed to provide a probabilistic prediction for an input or output event then this prediction will be about the current event.
- The modelling system needs to modify itself to take account of input or output events that have happened in reality or are being hypothetically regarded as having happened by the output vectoring system. If the modelling system is instructed to assume that a particular input or output event has happened with a particular value (e.g. that “0” or “1” has been input or output) then the input or output event that is being “fixed” in this way is the current event.

Changing the Current Event

The current event will need to change for two main reasons:

- Occurrence of real input and output events - As time passes input events or output events occur in reality. The modelling system needs to be told what value each of these has, so each becomes, in turn, the current event.
- Occurrence of hypothetical input and output events – As the output vectoring system simulates possible futures it needs to simulate possible future input and output events. If a tree search is used, for example, an input or output event would be simulated at each node of the search tree, as each node of a chess search tree simulates a move. Each of these simulated, or hypothetical, input and output events becomes, in turn, the current event.

Both of these situations – the real and the hypothetical – involve input and output events being observed by the modelling system, and used to update it, in chronological order. This means that when the current event is changed it will always change to the *next* input or output event (with one exception, relating to “undoing” in the recursive process, that will be discussed shortly). There is no need for any complex system for the modelling system to be told *which* input or output event is the next one. The modelling system merely needs to provide a service (*NextInputOutputEvent*) which allows it to be instructed to make the *next* input event or output event (chronologically) the current event.

Updating the Model

The modelling system observes the AI system’s inputs and outputs and makes probabilistic predictions of future inputs and outputs. An observation of an input or output involves the modelling system being told the value that was input or output for the input or output event that is happening, really or hypothetically, *now*. This is the “fixing” of an input or output bit. After being told this the modelling system modifies itself so that the predictions that it will produce for other, future, input and output events are altered accordingly. We do not need to know how this works: it is only an issue in the design of the modelling system itself.

For this to happen the modelling system needs a single service (*ApplyInputOutputEventToModel*) that allows it to be told what the value of the input received or the output made is for the current event so that it can update itself accordingly. A second service (*DoInputOutputEvent*) is actually provided which combines this with prioritization control (see below).

Obtaining Information About the Current Event

Within the output vectoring system, probabilistic predictions need to be obtained from the model for future input or output events. This requires the modelling system to have a service that, given some particular input or output value, gives the probability that the current input or output event will be found to have that value. For example, we may want to know the probability that the current input or output event has a value of “0” or we may want to know the probability that it has a value of “1”.

Clearly, if the inputs and outputs are all binary then a service which just returned the probability of the “1” input or output would be adequate – the probability for “0” being obtained by subtracting it from 1. In the example code here I have defined a service that returns the probability of either the “1” or “0” input or output, so as to avoid any difference in how particular values are treated.

As well as probabilities, other information needs to be obtained about the current input or output event:

- A service (*IsOutputEvent*) is needed to indicate whether the current event is an input event or output event. Another service (*IsInputEvent*) is also provided for completeness.
- Some outputs are special prioritization control outputs, meaning that they are used to control prioritization of computing resources in the model itself, rather than being sent to the outside world. A service (*IsPrioritizationControl*) is needed to indicate, in situations

where the current event is an output event, whether or not it is a prioritization control output event.

Undoing

When the output vectoring system is investigating possible future inputs and outputs the model is put into corresponding states reached by updating it with various sequences of hypothetical, future inputs and outputs. These updates are only wanted while performing that part of the search and need to be undone later.

The modelling system needs a service to allow undoing of changes that have recently been made to it. I have provided two such services in the example software here:

- A service (*UndoApplyInputOutputToModel*) which undoes the last update of the value for the current input or output event in the model.
- A service (*UndoNextInputOutputEvent*) which undoes the last changing of the current event – equivalent to making the previous event (chronologically) the current event again.

An alternative to an undo facility would be to make a *copy* of the modelling system each time it needed updating, and then when this copy needed updating to make a copy of it and so on. “Going back” would now just mean discarding the copy currently in use and reverting to the previous version of the model. This would be like a chess search tree passing a copy of the current state of the chess board to the next level of recursion as a value parameter. While this would work in principle, it could be wasteful of computing resources and would be a questionable practice, but anyone who really wanted to use the modelling system in this way could do so whether an undo facility was included or not. Nothing is gained by omitting an undo facility from the modelling system.

Evaluating

Planning as modelling needs an evaluation function. In the example software here I have defined this as a service (*SituationalEvaluationFunction*) of the modelling system, which is in a similar way to situational evaluation functions in chess algorithms [12]. It gives a score indicating the desirability of the model in any given state.

Inputs and Outputs

Services are needed to allow the AI system to interact with the outside world.

- A service (*GetActualInput*) is needed to obtain the next input due from the outside world. This should relate to the current event.
- A service (*MakeActualOutput*) is needed to send an output to the outside world with some specific value. This should also relate to the current event.

For now, I have included these services as part of the modelling system, even though they are not, technically, modelling activities. This is because they rely on information specific to the model.

Applying Prioritization Control Outputs

Planning as modelling uses prioritization control outputs – special pseudo-outputs that are sent back into the AI system and act as instructions to control prioritization of computing resources in the modelling system. Normal outputs are only applied when the system is responding to real inputs and are not applied when the output vectoring system is simulating possible futures, while prioritization control outputs, when they occur, are always applied. Furthermore, the effects of prioritization control outputs need to be undone when they are used hypothetically while the idea of undoing real outputs makes no sense. Prioritization control outputs are therefore not dealt with as normal application of outputs in the software provided here.

Prioritization control outputs are instead dealt with separately by another service (*ApplyPrioritizationControlToModel*). It is convenient to apply prioritization control outputs at the same time as updating the model because both updates of the model and prioritization control involve changes to the model that are performed whether the situation is a hypothetical one or not. Both prioritization control outputs and updates to the model may need to be undone. Because there is such a close link between model updates and application of prioritization control outputs, I have also provided a service (*DoInputOutputEvent*) in the software which combines updating of the model with application of prioritization control outputs so that a single command can be used to tell the model that an output event has occurred with a particular value and to apply that output to the model as a prioritization control output.

An alternative approach could have been used in software design: I could have dealt with the prioritization control outputs as very similar to real outputs (which is how they should conceptually be viewed), so that the service that makes outputs would also apply a prioritization control output, while only actually making the real output if the situation is not hypothetical.

The Two Versions of the Example Program

I have provided two examples of programs to demonstrate planning as modelling. The program source code listings are given in the appendices.

Standard Pascal Program in Appendix 1

The first example, in Appendix 1, is written in Standard Pascal. This program has the main control logic, modelling system and output vectoring system in a single source code file. The emphasis is on staying within Standard Pascal rather than good structuring of the system so that the program can be understood without knowledge of any specific variation of Pascal.

The model needs to be preserved throughout a run of the program: modelling system subroutines change a model which then remains in this state for other modelling system subroutines to extract information from it until later modelling subroutine calls make further changes to it. This makes Standard Pascal less than ideal for this purpose: variables in a Standard Pascal subroutine cannot maintain their states between different calls of the subroutine. There are solutions to this and, as the implementation of the modelling system is not shown here, no position is taken about which is used. One solution is for the model itself to be maintained in a separate program with which the

modelling system subroutines in this example interact. Another is for the model data to be maintained in a file. The most obvious solution, and an easy one to implement, is for the model to be represented by global variables. Global variables, however, are generally met with disapproval, as all of these methods will be by some readers.

Object Pascal/Delphi Programming Language in Appendix 2

Even when implemented in Standard Pascal, the system design is essentially one involving objects: the approach is already treating the model as an *object* with which there is interaction only by means of certain subroutines. An obvious approach is to use a variation of Pascal which allows explicit description of the model as something that maintains its state.

The second example, in Appendix 2, is written in the version of Pascal used by Borland's Delphi Pascal compiler. This is officially called *The Delphi Programming Language* by Borland but is sometimes known as *Object Pascal*. Object Pascal/The Delphi Programming Language provides object oriented programming facilities, which are used in this program to improve structure.

This second version of the program consists of three parts:

- **Main Program** – coordinates interaction of the modelling system and output vectoring system and uses the modelling system and output vectoring system units (below).
- **Modelling System Unit** – a separate unit providing the modelling system. The model is declared as an object class *TModel*, so when the main program needs to use a model it declares an instance of this class. Currently, only one such model is expected to be needed. The modelling system also declares methods for the model class to allow the model to be changed and information to be extracted from it from outside. This unit is used in the main program and the output vectoring system (see below).
- **Output Vectoring System Unit** – a separate unit providing the output vectoring system. The output vectoring system mainly consists of subroutines that perform a recursive tree search so it may seem that there is little need to declare the output vectoring system as an object class: the unit could just contain the subroutines. An output vectoring system object class *TOutputVectoringSystem* is declared, however, because there is one piece of data that is stored for the output vectoring system – the cut-off probability, which can be set from outside by using an output vectoring system method (*SetCutOffProbability*). Also, later developments in the output vectoring system may involve storage of more information from one instance of use to compute the value for an output to the next, to make the system more suitable for real-time use.

Components of the AI System

Functioning of the Main AI System

The main AI system coordinates interaction between the modelling system and output vectoring system.

The main program starts by setting up the model in its initial state – in the standard Pascal program by calling subroutine *InitializeModel*, in the Object Pascal/Delphi Programming Language program by using the TModel object's constructor method *Create*.

The main program is about to process real input and output events and undoing any of this makes no sense. After initializing the model, the program uses the modelling system's *DisableUndo* method/subroutine to indicate to the model that the facility to undo updates to the model is not required.

The program then processes each real input or output event in chronological order, as they actually occur in reality. The program repeatedly steps through the real input and output events using the modelling system's *NextInputOutputEvent* method/subroutine. Each call of this method/subroutine tells the modelling system that the main program has moved onto the next input event or output event, which should now become the current event.

For each input or output event the program needs to know whether it is an input event or an output event. This is done by using the modelling system's *IsOutputEvent* method/subroutine, which returns true if the current event is an output event and false if it is an input event.

If the current event is an input event then there is an actual input to collect from the outside world. This is done by the modelling system's *GetActualInput* method/subroutine. After the input is acquired then this input event needs to be "fixed" in the model – the modelling system needs to be told what value it had so that it can modify its predictions for future input events and output events accordingly. This "fixing" is done by using the modelling system's *DoInputOutputEvent* method/subroutine.

If the current event is an output event then the AI system needs to decide what output to make. This is the whole reason for use of planning as modelling. The system uses the output vectoring system's main method/subroutine *GetSelectedOutput* to determine the optimum output to make. This returns the optimum output – "0" or "1". This output will need to be made in the outside world, but only if it is not a prioritization control output: prioritization control outputs are special pseudo-outputs that only have an effect on the model itself. The program uses the modelling system's *IsPrioritizationControl* method/subroutine to check if the current event is a prioritization control output and only if this is not the case is the output sent to the outside world using the modelling system's *MakeActualOutput* method/subroutine. An important feature of planning as modelling is that the system's predictions are based on observation of its inputs and *its own outputs*. Therefore, regardless of whether this output is a prioritization control output or not, the modelling system is told what its value is – "0" or "1" – using the modelling system's *DoInputOutputEvent* method/subroutine – just as we do when dealing with an input. It is this method/subroutine that will also deal with the application of prioritization control, should this be a prioritization control output.

Notes:

This design does not deal with any real-time issues. It is assumed that the system never misses an input event or output event by taking too long on processing of previous events. One way of

imagining this is to assume that all input events are buffered and that there is no time pressure to make any outputs. In many real systems provision would need to be made for real-time requirements, for example by making the depth of the output vectoring system's search process dependent on available time or interrupting it and forcing it to give its best answer. This is an issue which can be resolved by conventional software engineering approaches.

The first action of the system after initializing the model is to disable undo for it. Because undo is disabled in the model's natural state, it would make more sense for the modelling system's *InitializeModel* subroutine or *Create* method to disable undo as part of its initialization of the model. I wanted to make it more obvious in these programs that this was happening and so put it in the main program. Undo is disabled in the main program, and not re-enabled at any point in the main program, so it may seem that undo is never enabled. In fact it is – just at a lower level – inside the output vectoring system's code when *GetSelectedOutput* is running.

Functioning of the Modelling System

The purpose of the modelling system is to analyse real or hypothetical previous inputs and outputs and make probabilistic predictions for future inputs and outputs.

InitializeModel – This subroutine is in the Standard Pascal program only. It sets the model up in its initial state.

Create - This is the Object Pascal/Delphi Programming Language program's equivalent of the *InitializeModel* subroutine. It is the constructor for the *TModel* object, which is the model. It sets the model up in its initial state.

EnableUndo – enables the undo facility. While it is enabled, any of the modelling system's methods/subroutines that make changes to the model can be reversed. This allows the model to be changed hypothetically to simulate possible futures in the output vectoring system.

DisableUndo – disables the undo facility. While it is disabled, any changes to the model are permanent.

NextInputOutputEvent – makes the next input event or output event (chronologically) after the current event the new current event.

EventProbability – returns the probability that the current event will occur with some given value – i.e. that when this event occurs, the AI system will receive an input with this value or make an output with this value. When the current event is an output event this method/subroutine involves the AI system in making a prediction about its own behaviour. Previous discussions of planning as modelling just had a single probability value for the input or output bit being “1” provided. This newer approach is better as it allows the software to be more easily altered later if inputs and outputs are needed that are not just single bits.

IsOutputEvent – returns true if and only if the current event is an output event. Both conventional outputs and prioritization control outputs will cause this method/subroutine to return true.

IsPrioritizationControl – returns true if and only if the current event is a prioritization control output event. Returns false if the current event is a conventional output event (i.e. not relating to prioritization control) or an input event.

UndoNextInputOutputEvent – undoes the last use of the *NextInputOutputEvent* method/subroutine or, if this has already been undone, the one before that, etc. i.e. makes the previous input event or output event (chronologically) the current event. Only valid when undo is enabled and when undo was enabled during the original use of *NextInputOutputEvent*.

ApplyInputOutputEventToModel – used to tell the modelling system that an input or output event has occurred and that the relevant input or output value should now be “fixed”. The event to which it relates is the current event, so this method/subroutine tells the modelling system what the the input or output value for the current event is. The modelling system updates itself so that any predictions for future input events or output events are altered accordingly. This method/subroutine can be used hypothetically, as its results can be reversed later (see below).

UndoApplyInputOutputEventToModel – undoes the last use of the *ApplyInputOutputEventToModel* method/subroutine or, if this has already been undone, the one before that, etc. Only valid when undo is enabled and when undo was enabled during the original use of *ApplyInputEventToModel*.

ApplyPrioritizationControlToModel – applies the current output to the model as a prioritization control output with a known value, i.e. the current event, with a given value is used as a command to control prioritization of computational resources in the modelling system. Only valid when the current event is a prioritization control output.

UndoApplyPrioritizationControlToModel – undoes the last use of the *ApplyPrioritizationControlToModel* method/subroutine or, if this has already been undone, the one before that, etc. Only valid when undo is enabled and was enabled during the original use of *ApplyPrioritizationControlToModel*.

SituationalEvaluationFunction – returns a score indicating the desirability of the situation described by the model. Intended to be used for hypothetical situations described by the model in the output vectoring system’s search of possible futures.

GetActualInput – retrieves the input from the outside world corresponding to the current event. Only valid when the current event is an input event. It is assumed that the next available input from the outside world *does* relate to the current event: the modelling system’s *NextInputOutputEvent* method/subroutine should ensure this.

MakeActualOutput – sends an output to the outside world corresponding to the current event. Only valid when the current event is an output event. It is assumed that the next expected output to the outside world *does* relate to the current event: the modelling system’s *NextInputOutputEvent* method/subroutine should ensure this. Prioritization control is not associated with this method/subroutine for reasons that I discussed earlier.

Extra Services:

The following services are not needed for the modelling system to be used, as their effects can be duplicated by using other services. They can, however, make the modelling system easier to use and are provided for convenience.

UndoEnabled – returns true if and only if the undo facility is enabled. Potentially useful in debugging.

IsInputEvent – returns true if and only if the current event is an input event. The opposite of **IsOutputEvent**.

DoInputOutputEvent – combines the functions of *ApplyInputOutputEventToModel* and *ApplyPrioritizationControlToModel*. It “fixes” the current event in the model with a given value, telling the modelling system that the input event or output event has occurred with that value and that the model should update itself accordingly. It also applies the current event to the model as a prioritization control output if the current event happens to be a prioritization control output. This method/subroutine can be used hypothetically as its effects can be undone. Although this method/subroutine is not required in the modelling system it is convenient and is used in the example programs in this article. For reasons discussed earlier, this method/subroutine associates prioritization control with model updating rather than, as you might reasonably expect, with the making of real outputs.

UndoInputOutputEvent – combines the functions of *UndoApplyInputOutputEventToModel* and *UndoApplyPrioritizationControlToModel*. Undoes the last use of the *DoInputOutputEvent* or, if this has already been undone, the one before that, etc. Only valid when undo is enabled and was enabled during the original use of *DoInputOutputEvent*. Although this method/subroutine is not required in the modelling system it is convenient and is used in the example programs in this article.

Functioning of the Output Vectoring System

The purpose of the output vectoring system is to use predictions of future inputs and outputs made by the modelling system to determine the optimum output value – in this case “0” or “1” – for a particular output event. The output vectoring system is therefore used only when there is an output to make.

The output vectoring system contains just the following methods/subroutines that are invoked from outside it:

GetSelectedOutput – the main method/subroutine in the output vectoring system. It is called when the current event is an output and the AI system needs a decision about which output to make – in this case “0” or “1”. Planning as modelling involves running simulations of possible future sequences of input events and output events, and the hypothetical changes to the model need to be undone. This method/subroutine therefore first enables the undo facility by using the modelling system’s *EnableUndo* method/subroutine. It starts the AI system’s planning as modelling recursive

tree search with a running probability of 1 by calling subroutine *TreeSearch*. This passes the optimum output back up so that it can be returned to the main program. Finally, undo is disabled with the modelling system's *DisableUndo* method/subroutine. The search process started by this method/subroutine is selectively limited in depth by the cut-off probability (see below).

SetCutOffProbability – only provided in the Object Pascal/Delphi Programming Language version of the program. The Standard Pascal program just uses a constant for the cut-off probability. Sets the cut-off probability to some given value. The cut-off probability is used to control prioritization of searches in the output vectoring system. The value should be higher when computing resources and time available to make decisions about outputs are higher.

GetCutOffProbability – only provided in the Object Pascal/Delphi Programming Language version of the program. Returns the current value of the cut-off probability set by the *SetCutOffProbability* method (above). This is not an essential method and is not used in the example programs but is provided because it may be convenient for debugging.

Output Vectoring System Internal Subroutines:

The following subroutines are internal to the output vectoring system. They are not used from outside it and are called, directly or indirectly, from the *GetSelectedOutput* method/subroutine.

TreeSearch – the subroutine that does the real work of planning as modelling. It is called for each new branch in the planning as modelling search tree. Each new branch corresponds to a decision about the value for a future input event or output event – in this case “0” or “1”. Each time the tree search is called a probability value – the running probability – is passed to it as a parameter. The running probability indicates the probability that the sequence of input events and output events described by the path through the search tree to the current branch will happen. The tree search can only continue as long as the running probability does not exceed the cut-off probability. The tree search starts two new branches of the search tree from the current node, each corresponding to a different value – “0” or “1” – for the current event. Each new branch is started by a separate call to the *GetLowerBranchScore* subroutine. Each time a new branch is started – for a particular value of some input or output event – the running probability is multiplied by the probability of that input or output event occurring with that value – obtained from the modelling system using the *EventProbability* method/subroutine. In this way the running probability decreases as the tree gets deeper, its value depending on how likely the future is described by the particular path through the search tree. When a particular path through the search tree is stopped – by the running probability being too low – the desirability of the hypothetical situation described by the model is determined by applying the modelling system's *SituationalEvaluationFunction* method/subroutine. This score is passed backed up the search tree. Each non-terminal node therefore receives two scores “backed up” from nodes below it. These need to be turned into a single score to be placed on that node and then passed further back up the tree. If the node is an output event the *GetScoreFromLowerOutputs* subroutine decides what score to place on this node. If the node is an input event the *GetScoreFromLowerInputs* subroutine decides what score to place on this node.

GetLowerBranchScore – causes a single branch of the search tree to be extended from the current node in the *TreeSearch* subroutine, corresponding to a particular value – “0” or “1” – occurring for

the input event or output event corresponding to the current node. This subroutine in turn calls *TreeSearch* (see above), resulting in indirect recursion.

GetScoreFromLowerOutputs – decides how to “back up” scores in the output vectoring system’s search tree when the current node corresponds to an output. Each node of the search tree corresponds to an input event or output event and has, in this case, two branches extending from it – for the “0” and “1” cases – to lower nodes, each of which will have a score on it. A single score needs to be “backed up” onto the current node. For an output event the system can be optimistic, because if it ever finds itself in the situation described by this node it can force the more desirable result by making the corresponding output. This subroutine therefore simply backs up the *larger* of the two sub-node scores.

GetScoreFromLowerInputs – fulfils a similar role to the *GetScoreFromLowerOutputs* subroutine (above), except that it backs scores up to an *input* event node. There is more potential for controversy here than for the output node case. I have opted, in this example, to back up the mean (expected) value, taking into account the probability of the branch leading to each sub-node being followed. For some tasks a more pessimistic, less risk-taking approach may be suitable.

Output Vectoring System Adjustment

Later versions of planning as modelling may feature some kind of adjustment of the output vectoring system. The output vectoring system currently prioritizes sequences of future input events and output events according to probability. By definition this causes a lot of prejudice in favour of outputs which lead to likely futures, which is the whole idea of constraint by the modelling system. If the modelling system were not being used to constrain the search then it would probably need to be restricted to sequences containing a certain future input and events: if a search tree were being used this would be a flat search tree. It may be that output vectoring would work better with less constraint from the modelling system, so that the system’s behaviour is a compromise between the output vectoring as described here and a “flat”, unconstrained search, making the system’s behaviour like that of the current system with all of the probability values from the modelling system “nudged” towards 0.5 to some degree (although there are other ways the degree of constraint could be moderated). It could, however, be preferable to *exaggerate* the constraint from the output vectoring system in some way.

Efficiency and Real-Time AI

The program listings here demonstrate an idea rather than a finished version. In practice, we would want to maximize the efficiency of the system and a range of techniques may be used to increase the efficiency of the output vectoring system’s search. Techniques for increasing the efficiency of tree searches are well known and will not be discussed here in detail. Many such techniques have been found in researching chess algorithms [13].

Although a tree search is used in this article, and that is how I usually think of it, planning as modelling does not demand a tree search: it merely demands that the output vectoring system perform a search for optimum behaviour constrained by future predictions of inputs and outputs.

Efficiency and management of resources may be a particular issue when the AI system must work in a real-time environment. The system described here treats reality like a chess game played without a clock, assuming that the world will wait while the system processes each input or output, before delivering the next input or accepting the next output. In reality, the system may need to limit its processing to keep within time constraints. The depth of the output vectoring system's tree search, if a tree search is used, may need to be controlled by varying the cut-off probability in accordance with available time. One simple way of achieving this could be first to search using a low cut-off probability, then a higher cut-off probability, then a still higher one and so on, until the system runs out of time and is forced to make an output based on what its last search found. A practice of performing repeated searches, increasing the search depth each time, is already used in chess programming.

An obvious way of increasing efficiency would be to partially reuse the results obtained by the output vectoring system's search for one output to reduce the computation needed for a slightly later output. This makes sense because if there is an output A and a slightly later output B then part of the future looked into during the output vectoring system's search for Output A will be part of the future which follows the making of Output B. The idea of using results obtained from the planning of one move to reduce the computation involved in planning of a later move is already known about in chess programming. There is the complication, when doing this with planning as modelling, that some of the decisions about backing up scores will be based on future input and output event probabilities which will have changed by the time the later output is performed but this could be dealt with.

It should be noted that techniques to increase efficiency and deal with real-time issues are likely to require amendment of the output vectoring system's code.

An obvious, though not necessarily correct, way of resolving real-time issues may be to use some of the prioritization control outputs to "tune" the output vectoring system, as well as controlling prioritization in the modelling system, so that the modelling system itself has some role in deciding how the output vectoring system is used. Another possibility is that there may be some demonstrably correct adjustment to make, or that it is not needed at all.

Conclusion

Planning as modelling uses a modelling system which makes probabilistic predictions to plan the future behaviour of an AI system. Modelling is made equivalent to planning. An important aspect of this is that the modelling system itself makes no distinction between the AI system's inputs and outputs: it observes both and makes predictions for both. This can be summarized as follows:

Planning is modelling.

This is an improvement on the system suggested by Hawkins [9,10], in which planning is still a discrete process, even if closely linked to modelling.

This article has provided example program code for crude AI systems using planning as modelling in Standard Pascal and Object Pascal/The Delphi Programming Language. These programs show

how the modelling system can be used to implement planning as modelling, but omit actual implementation of the modelling system itself. Methods/subroutines to provide the interface between the modelling system and the remainder of the AI system are, however, provided. The main purpose of the program listings is to demonstrate, unequivocally, one way in which planning as modelling could work.

Some readers may note that the program listings are short, considering that they are supposed to provide the basis on which an AI system can work. They would be even shorter if certain conventions had not been followed and they had not been structured to try to make them easy to understand and analyze. The programs should be expected to be short, however. All that intelligence does is model and plan. Planning as modelling suggests that planning can be made to occur in the modelling system, so it is appropriate that there should be hardly anything outside the modelling system. Ideally there would be *nothing* outside it. This is not completely attainable in practice: after all, without using any systems outside of the modelling system there is nothing we can actually do to implement planning as modelling, but we can get very close.

Although a separate system, the output vectoring system, is still needed outside the modelling system this is better than having a planning system outside the modelling system because the output vectoring system is not doing anything really profound and its implementation is more software engineering than real AI research. Modular design is possible for the modelling system and output vectoring systems. The output vectoring system does not need to “know” how the modelling system works. Output vectoring system designs do not need to be specific to models and reusable output vectoring system models can be produced. This allows successive, model independent output vectoring systems to be made, each more efficient than the last as knowledge is gained about increasing their efficiency, making very efficient output vectoring systems a realistic prospect. Likewise, a modelling system can be designed without knowledge of how the output vectoring system will work.

Adjustment of the output vectoring system could feature in later articles. It may be desirable to reduce the amount of constraint provided by the modelling system, making the system’s behaviour a compromise between the sort of search process described here and a “flat” search. It may, however, be preferable to exaggerate the constraint provided by the output vectoring system instead.

Issues of efficiency and real-time processing would need to be resolved, but these are of secondary importance and are likely to be conventional software engineering issues. The main requirement now to make this work as a simple AI system is a suitable probabilistic modelling system. How good this modelling system is will determine how well the AI system works. Prioritization control outputs are an important concept here, and although a modelling system could be made which does not use them, they would be needed for it to work very well.

References

[1] Web Reference: Almond, P. (2006). *Planning as Modelling in AI*. Retrieved 26 November 2006 from <http://www.paul-almond.com/PlanningAsModellingInAI.pdf>.

- [2] Web Reference: Almond, P. (2006). *How AI Would Work*. Retrieved 4 September 2006 from <http://www.paul-almond.com/HowAIWouldWork.pdf>.
- [3] Web Reference: Almond, P. (2006). *Occam's Razor Part 9: Representation and Planning of Actions in Artificial Intelligence*. Retrieved 29 July 2006 from <http://www.paul-almond.com/OccamsRazorPart09.pdf>.
- [4] Web Reference: Almond, P. (2006). AI as a Boundary System. Retrieved 17 September 2006 from <http://www.paul-almond.com/AIAsABoundarySystem.pdf>.
- [5] Web Reference: Almond, P. (2006). Occam's Razor Part 6: Partial Models as "Envelopes". Retrieved 1 March 2006 from <http://www.paul-almond.com/OccamsRazorPart06.htm>.
- [6] Web Reference: Almond, P. (2006). Occam's Razor Part 7: Hierarchy and Ontology. Retrieved 30 April 2006 from <http://www.paul-almond.com/OccamsRazorPart07.htm>.
- [7] Web Reference: Almond, P. (2006). Occam's Razor Part 8: Modelling in Artificial Intelligence. Retrieved 9 June 2006 from <http://www.paul-almond.com/OccamsRazorPart08.pdf>.
- [8] Web Reference: Almond, P. (2006). Downward Transfer of Probabilities in AI. Retrieved 15 October 2006 from <http://www.paul-almond.com/DownwardTransferOfProbabilitiesInAI.pdf>.
- [9] Hawkins, J., Blakeslee, S. (2004). *On Intelligence*. New York: Henry Holt.
- [10] Web Reference: George, D., Hawkins, J. (?). Belief Propagation and Wiring Length Optimization as Organizing Principles for Cortical Microcircuits. Retrieved 24 April 2006 from <http://www.stanford.edu/~dil/invariance/Download/CorticalCircuits.pdf>.
- [11] Levy, D.N.L. (1984). *The Chess Computer Handbook*. London: Batsford. Chapter 3, pp38-52.
- [12] Levy, D.N.L. (1984). *The Chess Computer Handbook*. London: Batsford. Chapter 2, pp7-37.
- [13] Heinz, E, A. (2000). *Scalable Search in Computer Chess: Algorithmic Enhancements and Experiments at High Search Depths*. Vieweg Verlag. Chapter 0, pp11-18.

Appendices

Appendix 1: Example Program in Standard Pascal

Standard Pascal Program Code

```
program AISystem1 (input, output);
{Illustrates the Planning as Modelling concept for AI.}
{Standard Pascal}
{AISystem1.pas}
{December 2006}
{www.paul-almond.com}
{Copyright Paul Almond 2006. All Rights Reserved}

const
    CutOffProbability = 1E-08; {An example value. If RunningProbability falls
below this value the current branch of the TreeSearch procedure cannot go
deeper.}
type
    TBit = 0..1; {binary digits}
var
    SelectedOutput: TBit; {the optimum output (0 or 1) found by the output
vectoring system for the next output event}

{MODELLING SYSTEM SUBROUTINES}

procedure InitializeModel;
{Puts the model in its initial state in which no previous inputs and outputs
have occurred.}
{Any probability values associated with future input or output events in the
model will be determined accordingly -}
{probably 0.5.}
begin
    {The code here is specific to the modelling system.}
end; {end of InitializeModel procedure}

procedure EnableUndo;
{Directs the modelling system to store enough information to allow any changes
to it to be reversed.}
{Allows changes to the modelling system by the output vectoring system's tree
search to be hypothetical.}
begin
    {The code here is specific to the modelling system.}
end; {end of EnableUndo procedure}

procedure DisableUndo;
{Directs the modelling system not to store enough information to allow changes
to be reversed and to discard any such information already stored.}
{Used when the model is being affected by real, rather than hypothetical, input
and output events.}
begin
    {The code here is specific to the modelling system.}
end; {end of DisableUndo procedure}

function UndoEnabled: boolean;
```

```

{Returns true if undo is enabled, otherwise returns false. Not used in this
program. Included for completeness and because it may be convenient for
debugging in a real system.}
begin
    {The code here is specific to the modelling system.}
end; {end of UndoEnabled function}

procedure NextInputOutputEvent;
{Makes the next input or output event (chronologically) after the current input
or output event the new current input or output event.}
begin
    {The code here is specific to the modelling system.}
end; {end of NextInputOutputEvent procedure}

function EventProbability (InputOutputValue:TBit): real;
{Returns the probability that a value of InputOutputValue (which is 0 or 1)
will be input or output when the current input or output event occurs.}
begin
    {The code here is specific to the modelling system.}
end; {end of EventProbability function}

function IsOutputEvent: boolean;
{Returns true if the current input or output event is an output event,
otherwise returns false.}
begin
    {The code here is specific to the modelling system.}
end; {end of IsOutputEvent function}

function IsInputEvent: boolean;
{Returns true if the current input or output event is an input event, otherwise
returns false.}
{Always returns the opposite of what IsOutputEvent would return.}
{Not used in this program and only included for completeness.}
begin
    IsInputEvent:= not IsOutputEvent;
end; {end of IsInputEvent function}

function IsPrioritizationControl: boolean;
{Returns true if the current input or output event is a prioritization control
output event (also implying an output event), otherwise returns false.}
begin
    {The code here is specific to the modelling system.}
end; {end of IsPrioritizationControl function}

procedure UndoNextInputOutputEvent;
{Undoes the effects of procedure NextInputOutputEvent, i.e. makes the previous
input or output event (chronologically) before the current input or output
event the new current input or output event.}
{Requires undo to be enabled.}
begin
    {The code here is specific to the modelling system.}
end; {end of UndoNextInputOutputEvent procedure}

procedure ApplyInputOutputEventToModel (InputOutputValue:TBit);
{Puts the model into the state that it should be in after the current input or
output event has occurred with an input or output of value InputOutputValue,
i.e. the relevant input or output bit is fixed.}

```

```

{This could be a real or hypothetical updating of the model.}
begin
    {The code here is specific to the modelling system.}
end; {end of ApplyInputOutputEventToModel procedure}

procedure UndoApplyInputOutputEventToModel;
{Undoes the effects of the last use of procedure ApplyInputOutputEventToModel.
Requires undo to be enabled.}
begin
    {The code here is specific to the modelling system.}
end; {end of UndoApplyInputOutputEventToModel procedure}

procedure ApplyPrioritizationControlToModel (OutputValue:TBit);
{Adjusts prioritization in the model, using the current input or output event
as a prioritization control output with value OutputValue.}
{Requires the current input output or output event to be a prioritization
control output.}
begin
    {The code here is specific to the modelling system.}
end; {end of ApplyPrioritizationControlToModel procedure}

procedure UndoApplyPrioritizationControlToModel;
{Undoes the effects of the last use of procedure
ApplyPrioritizationControlToModel.}
{Requires undo to be enabled.}
begin
    {The code here is specific to the modelling system.}
end; {end of UndoApplyPrioritizationControlToModel procedure}

procedure DoInputOutputEvent (InputOutputValue:TBit);
{Fixes the current input or output event as having occurred, actually or
hypothetically, with value InputOutputValue.}
{Updates the model accordingly so that any subsequent probabilities of input or
output events obtained from the model assume that the current input or output
event has occurred with value InputOutputValue.}
{If the input or output event is a prioritization control output then it
applies prioritization to the model.}
{Included for convenience. The effects of this procedure can be produced by
using procedures ApplyPrioritizationControlToModel and
ApplyInputOutputEventToModel.}
begin
    ApplyInputOutputEventToModel (InputOutputValue);
    if IsPrioritizationControl then
        ApplyPrioritizationControlToModel (InputOutputValue)
end; {end of DoInputOutputEvent procedure}

procedure UndoDoInputOutputEvent;
{Undoes the effects of the last use of procedure DoInputOutputEvent.}
{Requires undo to be enabled.}
{Included for convenience. The effects of this procedure can be produced by
using procedures UndoApplyPrioritizationControlToModel and
UndoApplyInputOutputEventToModel.}
begin
    if IsPrioritizationControl then
        UndoApplyPrioritizationControlToModel;
        UndoApplyInputOutputEventToModel
end; {end of UndoDoInputOutputEvent procedure}

```

```

function SituationalEvaluationFunction: real;
{Applies the situational evaluation function to the model and returns a score
indicating the desirability of the situation described by the model.}
begin
    {The code here is specific to the modelling system.}
end; {end of SituationalEvaluationFunction function}

{INPUT / OUTPUT SUBROUTINES}

procedure MakeActualOutput (OutputValue:TBit);
{Sends an actual output to the outside world. Has no effect on the model
itself.}
begin
    {The code here is specific to the modelling system.}
end; {end of MakeActualOutput procedure}

function GetActualInput: TBit;
{Receives an actual input from the outside world. Has no effect on the model
itself.}
begin
    {The code here is specific to the modelling system.}
end; {end of GetActualInput function}

{OUTPUT VECTORING SYSTEM SUBROUTINES}

procedure Treesearch (var SelectedOutput:TBit; var Score:real;
RunningProbability:real); forward;
{Forward declaration needed because of indirect recursion.}

function GetLowerBranchScore (InputOutputValue:TBit; RunningProbability:real):
real;
{Follows a branch of the search tree to a lower node, returning the score that
gets backed up to it.}
var
    NullOutput: TBit; {The selected output is of no interest at this level of
recursion.}
    Score: real; {the score backed up from lower levels of the search tree}
begin
    DoInputOutputEvent (InputOutputValue);
    NextInputOutputEvent;
    TreeSearch (NullOutput, Score, RunningProbability);
    UndoNextINputOutputEvent;
    UndoDoInputOutputEvent;
    GetLowerBranchScore:= Score
end; {end of GetLowerBranchScore function}

procedure GetScoreFromLowerOutputs (LowerScore0,LowerScore1:real; var
Score:real; var SelectedOutput:TBit);
{Decide what the score should be for this node, based on the two scores backed
up from branches to lower nodes for outputs of 0 and 1.}
{Also returns the output (0 or 1) associated with the selected score, though
this is only of interest in the top level of the tree search.}
begin
    {Select the higher score and the relevant output.}
    if LowerScore0 > LowerScore1 then
        begin

```

```

        Score:= LowerScore0;
        SelectedOutput:= 0
    end
    else
    begin
        Score:= LowerScore1;
        SelectedOutput:= 1
    end
end; {end of GetScoreFromLowerOutputs procedure}

function GetScoreFromLowerInputs
(LowerScore0,LowerScore1,Probability0,Probability1:real): real;
{Decides what the score should be for this node, based on the two scores backed
up from lower nodes for inputs of 0 and 1.}
begin
    {Determine the mean (expected) score}
    GetScoreFromLowerInputs:= (LowerScore0*Probability0 +
LowerScore1*Probability1) / 2;
end; {end of GetScoreFromLowerInputs function}

procedure Treesearch (var SelectedOutput:TBit; var Score:real;
RunningProbability:real);
{Searches recursively to determine the better output (0 or 1) for the current
output event.}
var
    LowerScore0, LowerScore1: real; {Scores backed up from the level below.}
begin

    {Has the cut-off limit been reached yet?}
    if RunningProbability < CutOffProbability then
    begin
        {Branch off for the 0 and 1 cases.}
        {Branch off for the 0 case, decreasing RunningProbability.}
        LowerScore0:= GetLowerBranchScore (0,
RunningProbability*EventProbability(0));
        {Branch off for the 1 case, decreasing RunningProbability.}
        LowerScore1:= GetLowerBranchScore (1,
RunningProbability*EventProbability(1));
        {Decide how to back up the score.}
        if IsOutputEvent then
            GetScoreFromLowerOutputs (LowerScore0, LowerScore1, Score,
SelectedOutput)
        else
            Score:= GetScoreFromLowerInputs (LowerScore0, LowerScore1,
EventProbability(0), EventProbability(1))
        end
    else
        {This is a terminal node. Return a score for it.}
        Score:= SituationalEvaluationFunction
    end; {end of TreeSearch procedure}

function GetSelectedOutput: TBit;
{The main output vectoring system routine. Starts the recursive search to
determine the better output (0 or 1) for the current output event.}
var
    SelectedOutput: TBit;

```

```

    NullScore: real; {the score associated with use of this output - not used
at this level of the program but included because the output vectoring system's
routine requires it as a variable parameter}
begin
    EnableUndo;
    TreeSearch (SelectedOutput, NullScore, 1);
    {1 is the running probability which will decrease with successive levels of
recursion depending on how likely the input or output branches are, until the
CutOffProbability threshold is reached.}
    DisableUndo;
    GetSelectedOutput:= SelectedOutput
end; {end of GetSelectedOutput procedure}

{MAIN PROGRAM}

begin
    InitializeModel;
    DisableUndo;
    {The normal state for the system, except when in an output vectoring system
tree search, is for the undo facility to be disabled.}
    {Process inputs and outputs in chronological order.}
    repeat
        NextInputOutputEvent;
        if IsOutputEvent then
            begin
                {Use output vectoring system to determine the better output.}
                SelectedOutput:= GetSelectedOutput;
                {Do the output for real.}
                if not IsPrioritizationControl then
                    MakeActualOutput (SelectedOutput);
                {Update the model.}
                DoInputOutputEvent (SelectedOutput)
            end
        else
            {Get the input value and use it to update the model.}
            DoInputOutputEvent (GetActualInput)
    until false {i.e. just keep running}
end. {end of AISystem1 program}

```

Appendix 2: Example Program in Object Pascal (Delphi Programming Language)

Main Program

```
program AISystem2 (input, output);
{Illustrates the Planning as Modelling concept for AI.}
{Object Pascal (Delphi Programming Language)}
{AISystem2.pas}
{December 2006}
{www.paul-almond.com}
{Copyright Paul Almond 2006. All Rights Reserved.}

uses Modelling, OutputVectoring;

const
    CutOffProbability = 1E-08; {An example value. If RunningProbability falls
below this value the current branch of the TreeSearch procedure cannot go
deeper.}

var
    Model: TModel;
    OutputVectoringSystem: TOutputVectoringSystem;
    SelectedOutput: TBit; {the optimum output (0 or 1) found by the output
vectoring system for the next output event}

begin
    {Initialize the modelling system.}
    Model := TModel.Create;
    Model.DisableUndo;
    {The normal state for the system, except when in an output vectoring system
tree search, is for the undo facility to be disabled.}
    {Initialize the output vectoring system.}
    OutputVectoringSystem := TOutputVectoringSystem.Create;
    OutputVectoringSystem.SetCutOffProbability (CutOffProbability);
    {If the running probability falls below this value the current branch of
the TreeSearch procedure cannot go deeper.}
    {Process inputs and outputs in chronological order.}
    repeat
        Model.NextInputOutputEvent;
        if Model.IsOutputEvent then
            begin
                {Use output vectoring system to determine the better output.}
                SelectedOutput := OutputVectoringSystem.GetSelectedOutput (Model);
                {Do the output for real.}
                if not Model.IsPrioritizationControl then
                    Model.MakeActualOutput (SelectedOutput);
                {Update the model.}
                Model.DoInputOutputEvent (SelectedOutput)
            end
        else
            {Get the input value and use it to update the model.}
            Model.DoInputOutputEvent (Model.GetActualInput)
    until false {i.e. just keep running}
end. {end of AISystem2 program}
```

Modelling System

```
Unit Modelling;
{Illustrates the Planning as Modelling concept for AI.}
{A modelling system which observes real or hypothetical inputs and outputs by
an AI system and provides probabilistic predictions of future inputs and
outputs.}
{Object Pascal (Delphi Programming Language)}
{Modelling.pas}
{December 2006}
{www.paul-almond.com}
{Copyright Paul Almond 2006. All Rights Reserved.}

interface

type

    TBit = 0..1;

    TModel = class

    private

        {Variable declarations specific to the modelling system go here.}
        {These are all internal to the modelling system - the only interaction
with code outside this unit is by using the methods defined here.}

    public

        constructor Create;
        {Creates an instance of the model and puts it in its initial state in which
no previous inputs and outputs have occurred.}

        procedure EnableUndo;
        {Directs the modelling system to store enough information to allow any
changes to it to be reversed.}
        {Allows changes to the modelling system by the output vectoring system's
tree search to be hypothetical.}

        procedure DisableUndo;
        {Directs the modelling system not to store enough information to allow
changes to be reversed and to discard any such information already stored.}
        {Used when the model is being affected by real, rather than hypothetical,
input and output events.}

        function UndoEnabled: boolean;
        {Returns true if undo is enabled, otherwise returns false. Included for
completeness and because it may be convenient for debugging in a real system.}

        procedure NextInputOutputEvent;
        {Makes the next input or output event (chronologically) after the current
input or output event the new current input or output event.}

        function EventProbability (InputOutputValue:TBit): real;
        {Returns the probability that a value of InputOutputValue (which is 0 or 1)
will be input or output when the current input or output event occurs.}
```

```

function IsOutputEvent: boolean;
{Returns true if the current input or output event is an output event,
otherwise returns false.}

function IsInputEvent: boolean;
{Returns true if the current input or output event is an input event,
otherwise returns false.}

function IsPrioritizationControl: boolean;
{Returns true if the current input or output event is a prioritization
control output event (also implying an output event), otherwise returns false.}

procedure UndoNextInputOutputEvent;
{Undoes the effects of procedure NextInputOutputEvent, i.e. makes the
previous input or output event (chronologically) before the current input or
output event the new current input or output event.}
{Requires undo to be enabled.}

procedure ApplyInputOutputEventToModel (InputOutputValue:TBit);
{Puts the model into the state that it should be in after the current input
or output event has occurred with an input or output of value InputOutputValue,
i.e. the relevant input or output bit is fixed.}
{This could be a real or hypothetical updating of the model.}

procedure UndoApplyInputOutputEventToModel;
{Undoes the effects of the last use of procedure
ApplyInputOutputEventToModel. Requires undo to be enabled.}

procedure ApplyPrioritizationControlToModel (OutputValue:TBit);
{Adjusts prioritization in the model, using the current input or output
event as a prioritization control output with value OutputValue.}
{Requires the current input output or output event to be a prioritization
control output.}

procedure UndoApplyPrioritizationControlToModel;
{Undoes the effects of the last use of procedure
ApplyPrioritizationControlToModel.}
{Requires undo to be enabled.}

procedure DoInputOutputEvent (InputOutputValue:TBit);
{Fixes the current input or output event as having occurred, actually or
hypothetically, with value InputOutputValue.}
{Updates the model accordingly so that any subsequent probabilities of
input or output events obtained from the model assume that the current input or
output event has occurred with value InputOutputValue.}
{If the input or output event is a prioritization control output then it
applies prioritization to the model.}
{Included for convenience. The effects of this procedure can be produced by
using procedures ApplyPrioritizationControlToModel and
ApplyInputOutputEventToModel.}

procedure UndoDoInputOutputEvent;
{Undoes the effects of the last use of procedure DoInputOutputEvent.}
{Requires undo to be enabled.}

```

{Included for convenience. The effects of this procedure can be produced by using procedures UndoApplyPrioritizationControlToModel and UndoApplyInputOutputEventToModel.}

```
function SituationalEvaluationFunction: real;
{Applies the situational evaluation function to the model and returns a
score indicating the desirability of the situation described by the model.}

{INPUT / OUTPUT SUBROUTINES}

procedure MakeActualOutput (OutputValue:TBit);
{Sends an actual output to the outside world. Has no effect on the model
itself.}

function GetActualInput: TBit;
{Receives an actual input from the outside world. Has no effect on the
model itself.}

end; {end of TModel class}
```

implementation

{MODELLING SYSTEM SUBROUTINES}

```
constructor TModel.Create;
{Creates an instance of the model and puts it in its initial state in which no
previous inputs and outputs have occurred.}
{Any probability values associated with future input or output events in the
model will be determined accordingly - probably 0.5.}
begin
{The code here is specific to the modelling system.}
end; {end of TModel.Create constructor}
```

```
procedure TModel.EnableUndo;
{Directs the modelling system to store enough information to allow any changes
to it to be reversed.}
{Allows changes to the modelling system by the output vectoring system's tree
search to be hypothetical.}
begin
{The code here is specific to the modelling system.}
end; {end of TModel.EnableUndo procedure}
```

```
procedure TModel.DisableUndo;
{Directs the modelling system not to store enough information to allow changes
to be reversed and to discard any such information already stored.}
{Used when the model is being affected by real, rather than hypothetical, input
and output events.}
begin
{The code here is specific to the modelling system.}
end; {end of TModel.DisableUndo procedure}
```

```
function TModel.UndoEnabled: boolean;
{Returns true if undo is enabled, otherwise returns false. Included for
completeness and because it may be convenient for debugging in a real system.}
begin
{The code here is specific to the modelling system.}
end; {end of TModel.UndoEnabled function}
```

```

procedure TModel.NextInputOutputEvent;
{Makes the next input or output event (chronologically) after the current input
or output event the new current input or output event.}
begin
    {The code here is specific to the modelling system.}
end; {end of TModel.NextInputOutputEvent procedure}

function TModel.EventProbability (InputOutputValue:TBit): real;
{Returns the probability that a value of InputOutputValue (which is 0 or 1)
will be input or output when the current input or output event occurs.}
begin
    {The code here is specific to the modelling system.}
end; {end of TModel.EventProbability function}

function TModel.IsOutputEvent: boolean;
{Returns true if the current input or output event is an output event,
otherwise returns false.}
begin
    {The code here is specific to the modelling system.}
end; {end of TModel.IsOutputEvent function}

function TModel.IsInputEvent: boolean;
{Returns true if the current input or output event is an input event, otherwise
returns false.}
{Always returns the opposite of what IsOutputEvent would return.}
{Not used in this program and only included for completeness.}
begin
    IsInputEvent:= not Self.IsOutputEvent;
end; {end of TModel.IsInputEvent function}

function TModel.IsPrioritizationControl: boolean;
{Returns true if the current input or output event is a prioritization control
output event (also implying an output event), otherwise returns false.}
begin
    {The code here is specific to the modelling system.}
end; {end of TModel.IsPrioritizationControl function}

procedure TModel.UndoNextInputOutputEvent;
{Undoes the effects of procedure NextInputOutputEvent, i.e. makes the previous
input or output event (chronologically) before the current input or output
event the new current input or output event.}
{Requires undo to be enabled.}
begin
    {The code here is specific to the modelling system.}
end; {end of TModel.UndoNextInputOutputEvent procedure}

procedure TModel.ApplyInputOutputEventToModel (InputOutputValue:TBit);
{Puts the model into the state that it should be in after the current input or
output event has occurred with an input or output of value InputOutputValue,
i.e. the relevant input or output bit is fixed.}
{This could be a real or hypothetical updating of the model.}
begin
    {The code here is specific to the modelling system.}
end; {end of TModel.ApplyInputOutputEventToModel procedure}

procedure TModel.UndoApplyInputOutputEventToModel;

```

```
{Undoes the effects of the last use of procedure ApplyInputOutputEventToModel.  
Requires undo to be enabled.}  
begin
```

```
    {The code here is specific to the modelling system.}
```

```
end; {end of TModel.UndoApplyInputOutputEventToModel procedure}
```

```
procedure TModel.ApplyPrioritizationControlToModel (OutputValue:TBit);  
{Adjusts prioritization in the model, using the current input or output event  
as a prioritization control output with value OutputValue.}  
{Requires the current input output or output event to be a prioritization  
control output.}
```

```
begin
```

```
    {The code here is specific to the modelling system.}
```

```
end; {end of TModel.ApplyPrioritizationControlToModel procedure}
```

```
procedure TModel.UndoApplyPrioritizationControlToModel;  
{Undoes the effects of the last use of procedure  
ApplyPrioritizationControlToModel.}
```

```
{Requires undo to be enabled.}
```

```
begin
```

```
    {The code here is specific to the modelling system.}
```

```
end; {end of TModel.UndoApplyPrioritizationControlToModel procedure}
```

```
procedure TModel.DoInputOutputEvent (inputOutputValue:TBit);
```

```
{Fixes the current input or output event as having occurred, actually or  
hypothetically, with value InputOutputValue.}
```

```
{Updates the model accordingly so that any subsequent probabilities of input or  
output events obtained from the model assume that the current input or output  
event has occurred with value InputOutputValue.}
```

```
{If the input or output event is a prioritization control output then it  
applies prioritization to the model.}
```

```
{Included for convenience. The effects of this procedure can be produced by  
using procedures ApplyPrioritizationControlToModel and  
ApplyInputOutputEventToModel.}
```

```
begin
```

```
    Self.ApplyInputOutputEventToModel (InputOutputValue);
```

```
    if Self.IsPrioritizationControl then
```

```
        Self.ApplyPrioritizationControlToModel (InputOutputValue)
```

```
end; {end of TModel.DoInputOutputEvent procedure}
```

```
procedure TModel.UndoDoInputOutputEvent;
```

```
{Undoes the effects of the last use of procedure DoInputOutputEvent.}
```

```
{Requires undo to be enabled.}
```

```
{Included for convenience. The effects of this procedure can be produced by  
using procedures UndoApplyPrioritizationControlToModel and  
UndoApplyInputOutputEventToModel.}
```

```
begin
```

```
    if Self.IsPrioritizationControl then
```

```
        Self.UndoApplyPrioritizationControlToModel;
```

```
        Self.UndoApplyInputOutputEventToModel
```

```
end; {end of TModel.UndoDoInputOutputEvent procedure}
```

```
function TModel.SituationalEvaluationFunction: real;
```

```
{Applies the situational evaluation function to the model and returns a score  
indicating the desirability of the situation described by the model.}
```

```
begin
```

```
    {The code here is specific to the modelling system.}
```

```

end; {end of TModel.SituationalEvaluationFunction function}

{INPUT / OUTPUT SUBROUTINES}

procedure TModel.MakeActualOutput (OutputValue:TBit);
{Sends an actual output to the outside world. Has no effect on the model
itself.}
begin
    {The code here is specific to the modelling system.}
end; {end of TModel.MakeActualOutput procedure}

function TModel.GetActualInput: TBit;
{Receives an actual input from the outside world. Has no effect on the model
itself.}
begin
    {The code here is specific to the modelling system.}
end; {end of TModel.GetActualInput function}

end. {end of ModellingSystem unit}

```

Output Vectoring System

```

Unit OutputVectoring;
{Illustrates the Planning as Modelling concept for AI.}
{Performs a look-ahead tree search to determine the optimum output. The search
is constrained by the modelling system.}
{This is not a planning system: the main planning is provided by the modelling
system in the form of the constraint.}
{Object Pascal (Delphi Programming Language)}
{OutputVectoring.pas}
{December 2006}
{www.paul-almond.com}
{Copyright Paul Almond 2006. All Rights Reserved}

interface

uses Modelling;

type

    TOutputVectoringSystem = class

    private

        {Variable declarations specific to the output system go here.}
        {These are all internal to the output vectoring system - the only
interaction with code outside this unit is by using the methods defined here.}

        CutOffProbability: real; {If the running probability falls below this value
the current branch of the TreeSearch procedure cannot go deeper.}

        {Currently the cut-off probability is the only data stored by the output
vectoring system, but other data could be added later.}

        function GetLowerBranchScore(Model:TModel; InputOutputValue:TBit;
RunningProbability:real): real;

```

```

    {Follows a branch of the search tree to a lower node, returning the score
that gets backed up to it.}

    procedure GetScoreFromLowerOutputs(LowerScore0,LowerScore1:real; var
Score:real; var SelectedOutput:TBit);
    {Decide what the score should be for this node, based on the two scores
backed up from branches to lower nodes for outputs of 0 and 1.}
    {Also returns the output (0 or 1) associated with the selected score,
though this is only of interest in the top level of the tree search.}

    function GetScoreFromLowerInputs
(LowerScore0,LowerScore1,Probability0,Probability1:real): real;
    {Decides what the score should be for this node, based on the two scores
backed up from lower nodes for inputs of 0 and 1.}

    procedure TreeSearch (Model:TModel; var SelectedOutput:TBit; var
Score:real; RunningProbability:real);

    public

    procedure SetCutOffProbability (NewCutOffProbability:real);
    {Sets the cut-off probability to NewCutOffProbability.}
    {If the running probability falls below this value the current branch of
the TreeSearch procedure cannot go deeper.}

    function GetCutOffProbability: real;
    {Returns the current cut-off probability which was set by procedure
SetCutOffProbability. Included for convenience and possible use in debugging.}

    function GetSelectedOutput (Model:TModel): TBit;
    {The main output vectoring system routine. Starts the recursive search to
determine the better output (0 or 1) for the current output event.}

end; {end of TOutputVectoringSystem class}

implementation

procedure TOutputVectoringSystem.SetCutOffProbability
(NewCutOffProbability:real);
{Sets the cut-off probability to NewCutOffProbability.}
{If the running probability falls below this value the current branch of the
TreeSearch procedure cannot go deeper.}
begin
    Self.CutOffProbability:= NewCutOffProbability;
end; {end of TOutputVectoringSystem.SetCutOffProbability procedure}

function TOutputVectoringSystem.GetCutOffProbability: real;
{Returns the current cut-off probability which was set by procedure
SetCutOffProbability. Included for convenience and possible use in debugging.}
begin
    GetCutOffProbability:= Self.CutOffProbability;
end; {end of TOutputVectoringSystem.GetCutOffProbability function}

function TOutputVectoringSystem.GetLowerBranchScore (Model:TModel;
InputOutputValue:TBit; RunningProbability:real): real;
{Follows a branch of the search tree to a lower node, returning the score that
gets backed up to it.}

```

```

var
  NullOutput: TBit; {The selected output is of no interest at this level of
recursion.}
  Score: real; {the score backed up from lower levels of the search tree}
begin
  Model.DoInputOutputEvent (InputOutputValue);
  Model.NextInputOutputEvent;
  Self.TreeSearch (Model, NullOutput, Score, RunningProbability);
  Model.UndoNextInputOutputEvent;
  Model.UndoDoInputOutputEvent;
  GetLowerBranchScore:= Score
end; {end of TOutputVectoringSystem.GetLowerBranchScore function}

procedure TOutputVectoringSystem.GetScoreFromLowerOutputs
(LowerScore0,LowerScore1:real; var Score:real; var SelectedOutput:TBit);
{Decide what the score should be for this node, based on the two scores backed
up from branches to lower nodes for outputs of 0 and 1.}
{Also returns the output (0 or 1) associated with the selected score, though
this is only of interest in the top level of the tree search.}
begin
  {Select the higher score and the relevant output.}
  if LowerScore0 > LowerScore1 then
  begin
    Score:= LowerScore0;
    SelectedOutput:= 0
  end
  else
  begin
    Score:= LowerScore1;
    SelectedOutput:= 1
  end
end; {end of TOutputVectoringSystem.GetScoreFromLowerOutputs procedure}

function TOutputVectoringSystem.GetScoreFromLowerInputs
(LowerScore0,LowerScore1,Probability0,Probability1:real): real;
{Decides what the score should be for this node, based on the two scores backed
up from lower nodes for inputs of 0 and 1.}
begin
  {Determine the mean (expected) score}
  GetScoreFromLowerInputs:= (LowerScore0*Probability0 +
LowerScore1*Probability1) / 2;
end; {end of TOutputVectoringSystem.GetScoreFromLowerInputs function}

procedure TOutputVectoringSystem.TreeSearch (Model:TModel; var
SelectedOutput:TBit; var Score:real; RunningProbability:real);
{Searches recursively to determine the better output (0 or 1) for the current
output event.}
var
  LowerScore0, LowerScore1: real; {Scores backed up from the level below.}
begin
  {Has the cut-off limit been reached yet?}
  if RunningProbability < Self.CutOffProbability then
  begin
    {Branch off for the 0 and 1 cases.}
    {Branch off for the 0 case, decreasing RunningProbability.}

```

```

        LowerScore0:= Self.GetLowerBranchScore (Model, 0,
RunningProbability*Model.EventProbability(0));
        {Branch off for the 1 case, decreasing RunningProbability.}
        LowerScore1:= Self.GetLowerBranchScore (Model, 1,
RunningProbability*Model.EventProbability(1));
        {Decide how to back up the score.}
        if Model.IsOutputEvent then
            Self.GetScoreFromLowerOutputs (LowerScore0, LowerScore1, Score,
SelectedOutput)
        else
            Score:= Self.GetScoreFromLowerInputs (LowerScore0, LowerScore1,
Model.EventProbability(0), Model.EventProbability(1))
        end
        else
            {This is a terminal node. Return a score for it.}
            Score:= Model.SituationalEvaluationFunction
end; {end of TOutputVectoringSystem.TreeSearch procedure}

function TOutputVectoringSystem.GetSelectedOutput (Model:TModel): TBit;
{The main output vectoring system routine. Starts the recursive search to
determine the better output (0 or 1) for the current output event.}
var
    SelectedOutput: TBit;
    NullScore: real; {the score associated with use of this output - not used
at this level of the program but included because the output vectoring system's
routine requires it as a variable parameter}
begin
    Model.EnableUndo;
    Self.TreeSearch (Model, SelectedOutput, NullScore, 1);
    {1 is the running probability which will decrease with successive levels of
recursion depending on how likely the input or output branches are, until the
CutOffProbability threshold is reached.}
    Model.DisableUndo;
    GetSelectedOutput:= SelectedOutput
end; {end of TOutputVectoringSystem.GetSelectedOutput procedure}

end. {end of OutputVectoringSystem unit}

```