The INFACT GMA Gnome: Providing Automated Student Feedback via Graphical Models

Nathan Evans
Advised by Steve Tanimoto

Senior Honors Thesis
June 2005
Department of Computer Science and Engineering
University of Washington

1 Introduction

In an effort to enhance the educational process, we can utilize computational multitasking and processing power via online learning environments to effectively monitor students' progress and diagnose individual cognitive states. Serving as an aid to both educators and students, the Graphical Model Assessment Gnome (GMA Gnome) aims to provide student-specific automated feedback. For teachers, this feedback can be used to differentiate those students who are struggling conceptually with certain material from those who have a firm grasp of the material. Additionally, it can serve as a meta-level indication of whether or not the teacher is effectively presenting the information to the pupils. Students in turn benefit from automated, individualized feedback in classroom situations where a single teacher may not be able to provide individual attention at all times. Figure 1.1 schematically shows the relationships among the student, the GMA Gnome, and the teacher.

The GMA Gnome is a server-side software agent that listens and processes all historical and real-time input made by students in an online learning environment. Sampling input of varying data types across multiple pieces of software, the GMA Gnome combines and processes incoming events using a specialized graphical model and probabilistic reasoning to determine whether or not to perform a specified action. These actions can act upon a cohesive prediction of an individual student's cognition and progress at a given task or concept.

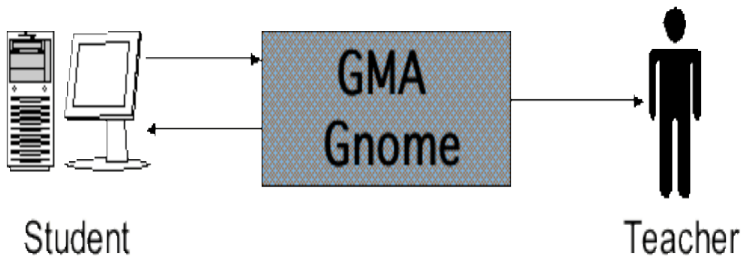

Figure 1.1
The GMA Gnome: An Educational Tool for Students and Teachers

2 Contextual Background

The GMA Gnome has been developed on a large bed of foundational systems. Explanation of the history, purpose, and utility of these systems will hopefully shed light on the scope and context of the GMA Gnome project.

2.1 Online Learning Environments

The increasing ubiquity and advancement of computing has definite potential to enhance and influence the educational realm. One method of integrating computers into education is to embed the entire class into an online learning environment. That is, provide to students an integrated, networked, computer-based environment in which material presentation, communication, and student construction are key players. Online learning environments have several educational benefits including: the ability to easily interact with class material, convenience of access, ease of collaboration, and extensibility to provide assessment.

2.2 Educational Assessment

Providing some form of assessment is common practice in education. Assessment generally comes in two flavors: summative assessment and formative assessment. Summative assessment involves measuring student understanding and learning after a sustained period of instruction. This form of assessment usually translates into providing a grade for students. Formative assessment, on the other hand, provides information and feedback to students throughout the course of the learning process in an effort to continually tailor and enhance the learning process for students.

By utilizing technology we can implement systems of assessment that require little or no human interaction to function. Such systems are generally referred to as automated assessment systems. An idealized automated assessment system involves a powerful machine autonomously providing useful, individuated, real-time feedback to students in an online learning environment. Automated assessment offers direct benefits to both students and teachers. Students can receive timely feedback in situations where the teacher may be unable to grant all students individual attention. Teachers, in turn, obtain greater productivity: they can now spend their time supplementing the automated assessment instead of having to personally supply it.

Our research focuses on providing unobtrusive educational assessment that utilizes generated student-activity data while students continue to work. Unobtrusive assessment is performed latently and does not disrupt its recipients.

2.2.1 Automated Assessment: Approaches in Online Learning Environments

It is necessary to select a specific assessment approach to best fit the desired effects of providing automated assessment in an online learning environment. One approach is to employ a multiple-choice system. Multiple-choice systems are a natural way to implement summative assessment as they can easily supply a measurement of student understanding by means of a grade. Since there are a discrete set of answers that students can respond to, providing automated assessment via multiple-choice can be implemented with ease. One caveat with multiple-choice systems is that assessment benefits are obscured to the students. When students merely receive a score it is hard for them to learn much from the assessment. For instance, they may be unable to ascertain the reason they missed a question and will be unable to identify and correct any misconceptions they may hold with respect to the material.

A second form of automated assessment is to perform textual analysis. The idea behind automated textual analysis is to extract the meaningful semantic characteristics in a large body of text and to provide assessment based on the student's underlying understanding. One common form of automated textual analysis is automated essay grading. Extracting the semantics from a body of text can be done using a myriad of natural language processing techniques including Latent Semantic Analysis and statistical use of syntactic features such as word frequency counts. Unlike multiple-choice systems, students submitting a body of text do not have a finite set of possible answers to select from. This makes performing textual analysis a particularly difficult task as a means for implementing reliable and meaningful automated assessment.

A final approach is to provide automated assessment as a fusion of multiple forms of evidence. Instead of focusing on one specific form of input such as multiple-choice input, textual input, or UI events, automated assessment of this form is performed with respect to the combination of multiple forms of input. Observing evidence from multiple forms of input requires a way of meaningfully combining the evidence from disparate sources. One such method is to use probabilistic inference. Evidence fusion can be instrumental in providing a more comprehensive picture of student understanding since it does not limit assessment generated from only one form of input. I will be focusing on this approach for the remainder of the paper since the GMA Gnome provides automated assessment by combining various forms of evidence using probabilistic inference.

2.3 The INFACT Online Learning Environment

The GMA Gnome has been developed to specifically interact with the **I**ntegrated **N**etworked **F**acet-based **A**ssessment **C**apture **T**ool (INFACT) online learning environment, a collaborative research project involving various departments at the University of Washington. Describing the purpose, intent, and content of the INFACT project will help motivate and provide context to the GMA Gnome.

Assessment within the INFACT environment is done by means of a facet-based methodology created by Jim Minstrell [1] in which student understanding is quantified in terms of a set of *facets*, or common conceptions and misconceptions. The INFACT project has been developed as a facet-based educational platform with three primary goals in mind. First, it is desired to have a collaborative environment in which students can work together and learn from one another. Second, the environment must offer tools whereby students can interactively learn, participate, and construct. Finally, the environment needs to include mechanisms for performing unobtrusive, automated assessment.

INFACT additionally takes advantage of technological capabilities by recording and storing all input generated from students in the online learning environment. Complete storage of all inter-environmental events makes available to educational researchers an entire trace of students' conceptual understanding of course material. This extensively stored data can later be analyzed by observing trends in the students' facet distributions. Analysis can help teachers customize curriculum to better match the observed needs of students.

At the core of the INFACT environment lie four general components: the forum, student tools, teacher tools, and educational researcher tools. The forum is a standard online forum-posting system that contains groups of users such as students or teachers who can post and reply to messages. The cluster of student tools includes course-specific software designed to interact with INFACT by storing all inter-software events to the shared INFACT databases. Examples of student tools include the PixelMath (see section 2.3.1), the Sketch Tool, and IPAQ games. Teacher tools include software to visualize student event streams, administer student accounts and privileges, and to provide markup assessment. The final suite of INFACT components is geared toward educational researchers and includes facetbases populated with facets about a given subject, data visualization tools, and automated assessment mechanisms.

Given the complexity of the INFACT environment, the GMA Gnome aims to cohesively combine the various potential sources of incoming student data into predicted states of student understanding. Moreover, the GMA Gnome bridges all basic INFACT components in that it runs on models created by educational researchers, can be implemented by teachers, and provides feedback to students via the forum.

2.3.1 PixelMath: An INFACT Student Tool

INFACT provides a modular online learning environment in which specific course material and can be taught by plugging into INFACT specialized course software. One subject area that has been tailored to be taught via INFACT is image processing. Students perform image processing and calculating in INFACT by means of a student tool called PixelMath that conforms to the INFACT model of logging all student events to the INFACT database [2].

PixelMath allows students to open digital images in order to select and manipulate pixels. It provides features for selecting pixels, zooming in and out on pixels, and performing pixel manipulations via pixel calculation. Throughout the remainder of this paper I will be referring to PixelMath and its functions for demonstrative purposes.

2.4 The Gnome Environment

The Gnome Environment [3] is an extension of INFACT that provides a platform for development and deployment of transparent autonomous agents called *Gnomes*. The Gnome Environment model is constructed such that there exists a pool of concurrently running Gnomes, each listening to all incoming events within INFACT and performing tasks given those events. Gnomes are useful for a variety of tasks, including, but not limited to monitoring INFACT logins, determining a student's "time on task," or providing automated facet-based assessment (as the GMA Gnome does). The GMA Gnome has been developed using the Gnome Environment and is one of the autonomous agents in the Gnome pool.

When an event is generated within INFACT, it is simultaneously forwarded to the *Event Dispatcher* and recorded to the *INFACT database* (see Figure 2.1). At this point, the event is dispatched to the pool of Gnomes (including the GMA Gnome) for all of the Gnomes to sample. If a Gnome is programmed to respond to such an event, it has two options. It can either request or write information to the *INFACT database* via the *Data Interface* or it can make an external response via e-mail or by posting to the forum.

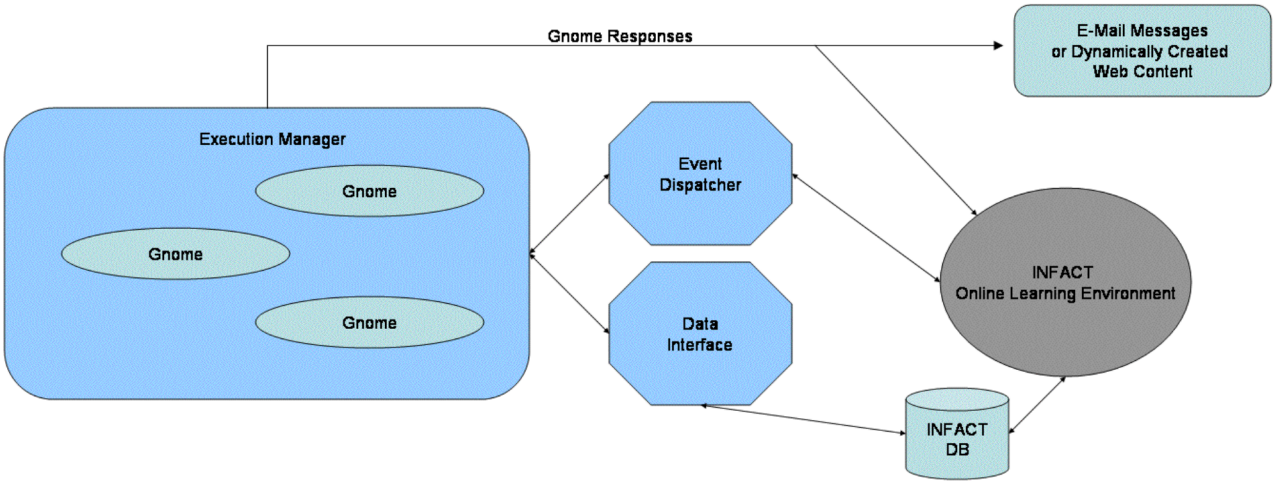

Figure 2.1
Architectural Outline of the Gnome Environment **[3]**

3 Theoretical Foundations of the GMA Gnome

In order to make reasonable diagnoses of a particular student's state of understanding on which actions are to be based, we have chosen to employ techniques of artificial intelligence. To effectively trace transitions in the state of student cognition or progress through an activity, the GMA Gnome bases its diagnoses on sequential student input in INFACT and utilizes a specialized form of a graphical model.

3.1 Graphical Models

Graphical models are a unified approach to understanding and developing algorithms across various methodological domains. A graphical model symbolizes dependencies among random variables and a factorization of the joint probability distribution of all of the random variables. Each node in a graphical model represents a random variable and arcs, or edges, indicate dependencies among the random variables. Random variables (nodes) in a graphical model that lack ancestral or descendent relationships with other nodes are defined as being conditionally independent given the values of their parents. Given particular evidence, graphical models provide us with a mechanism to update inter-network probabilities to arrive at a probability that a hypothesis is true or false. Examples of graphical models include Bayesian (belief) networks, neural nets, and hidden Markov models.

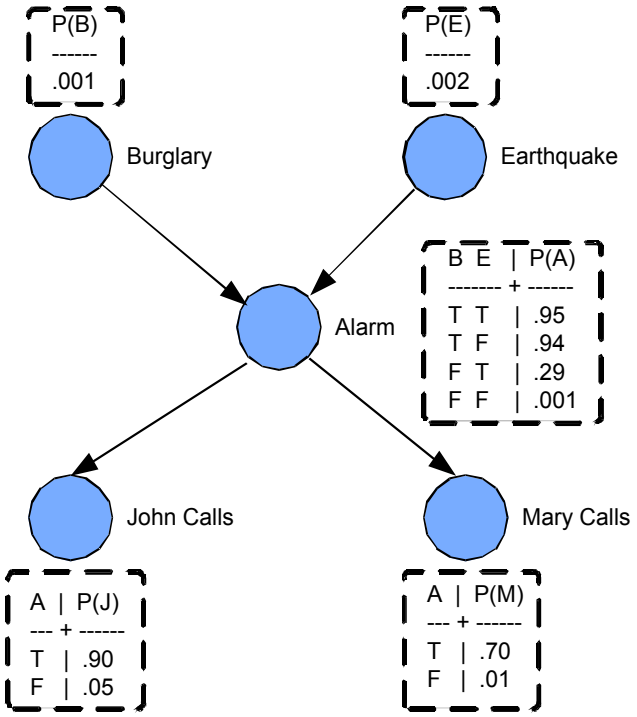

Figure 3.1
Earthquake Bayesian Network [4]

3.1.1 Bayesian Network Example

Stepping through an example graphical model will be instructive in introducing the vocabulary and functions of graphical models. Figure 3.1 is a graphical representation of a Bayesian network modeled around the following scenario. Imagine that your house in Los Angeles has an alarm system attached to it. Also, imagine that you have two neighbors named John and Mary. You have asked John and Mary to call you at work if your alarm system goes off. Unfortunately, your alarm system is not purely sensitive to just burglaries. Past Los Angeles earthquakes have set off the alarm, and if another earthquake were to happen, it too may trigger the alarm.

The above network contains five random variables: burglary occurring, earthquake occurring, alarm sounding, John calling, and Mary calling. If we observe any of these random variable events to actually occur, we consider it to be *evidence*. Any time that we witness evidence of some random variable it is *triggered* and *fires* to its *children nodes* (downstream nodes). The process of firing involves propagating a *posterior probability* to its children who use a *conditional probability table* to determine their own posterior probability and then continue to propagate their values downstream. The conditional probability table is an exhaustive mapping of the truth-values of all parents of a node to an associated posterior probability.

In particular, probabilities within graphical models are mathematically defined as follows [4]:

If the events are:
$X_1 \ldots X_n$

Then the joint probability:
$P(X_1 \ldots X_n)$

Is equal to the product of the conditional probabilities:
$P(X_i \mid \text{parents of } X_i) \text{ for } i = 1,\ldots,n$

3.2 Motivating a Specialized Graphical Model

When choosing how to model student understanding and task progress, there are a number of features that will simplify the process. In short, we desire an intuitive, powerful model that recognizes and combines multiple forms of evidence. As outlined below, traditional graphical models are limited and lack essential features helpful in providing automated assessment within online learning environments.

Standard graphical models externalize evidence capturing. That is, nodes within typical graphical models are not defined by mechanisms to distinguish evidence and respond accordingly. A model-defined, internalized representation of

evidence distinction grants our model additional power in that it is self-sufficient with respect to external, incoming events.

A second shortcoming of typical graphical models is that they contain no manner with which to accumulate evidence. When a piece of evidence occurs, it merely triggers the evidence node once. The ability to detect subsequent instances of identical evidence is important in educational assessment because it may well be the case that repeated evidence is indicative of an increase in student understanding.

Another important characteristic that an intuitive, powerful model ought have is the ability to recognize event sequence ordering. Standard graphical models include no way of distinguishing event order; that is, though you can ascertain if event A and event B have happened, there is no way to tell which happened first. When performing sequential tasks in a student activity, student understanding may hinge upon a particular order of input events.

Finally, standard graphical models externalize actions. Nodes within a graphical model are not generally defined to perform any actions if their posterior probability exceeds some threshold. Exceeding the threshold in a terminal node simply indicates the probability of that hypothesis actually being true given the seen evidence. Without the intrinsic ability to perform actions, the assessment model would lack an interactive apparatus.

With respect to educational assessment, these four limitations of standard graphical models motivate the specialized, feedforward graphical model that we have created. Our feedforward graphical model includes in its definition a way to capture and filter evidence, accumulate evidence, recognize structural event order, and perform actions in addition to basic probability propagation. I will now introduce the basic components and the relevant features of our specialized models.

3.3 Specialized Graphical Models

The specialized graphical model is structurally identical to other feedforward graphical models. We endow each node a prior probability, conditional probability table, and a posterior probability. We define a hierarchical model with three types of nodes: input, middle, and output. Input nodes are responsible for capturing specific evidence, collecting repeated evidence, and recognizing event sequence. Middle nodes serve as evidence aggregators. Output nodes contain a set of associated actions and a threshold that, if exceeded, indicates that its associated actions must be performed.

3.3.1 Input Nodes

As mentioned above, the main function of an input node is to recognize and gather evidence. Input nodes are the first and most complex layer of our

hierarchical model. Input nodes constantly listen to incoming events and test them against defined filters. Input nodes are defined in part by an extended condition language that fully specifies what evidence a given input node ought be sensitive to (see Appendix B). An input node may contain an unlimited number of conditional filters to selectively isolate a particular type of event. The conditional filters are considered as a short-circuited conjunction; namely, as soon as one condition goes unmatched, the input node ignores the evidence.

Events are primarily dispatched as a string description of the event that occurred. As such, evidence recognition for input nodes is primarily textually based. The two main mechanisms for pattern matching an event string are to use either regular expressions or an extension of regular expressions. The extended regular expression pattern matching is a shortcut to include numerical ranges within standard regular expressions (see Appendix B).

Input nodes contain two constructs with which they can accumulate evidence. Each alters the behavior of an input node with respect to when and how it fires (i.e. invokes an update in all downstream nodes). A third mode (ONE_SHOT) does not collect evidence. Evidence handling methods are defined as follows:

- REQUIRE (*N):*

  The REQUIRE statement demands that an input node successfully match and recognize *N* instances of evidence before firing to its downstream nodes. This mode is particularly useful if a certain number of evidence instances is meaningful in modeling some aspect of student understanding. It should be noted that standard graphical models are simply a special case of the REQUIRE mode with N=1.

- ACCUM_ZENO (*X*):

  This mode is based on Zeno's dichotomy paradox that states to get from point A to point B you must first go halfway. Then, to get from the halfway mark to B you must again go halfway. Zeno pointed out that if one continually has to go halfway between their current position and B, they would never end up at point B!

  One limitation of the REQUIRE construct is that it will not fire until it accumulates enough evidence to justify a fire. ACCUM_ZENO is a way to model the case where a student is assumed to have a vague understanding the first time they provide evidence and an increasing understanding with every subsequent piece of identical evidence. The parameter *X* represents a fraction of the distance that we ought to increase the posterior with every subsequent instance of identical evidence. For example, if *X*=0.5, the posterior probability of the input node will be increased to half of the distance from its current position to 1.0.

- ONE_SHOT:
  Input nodes defined to be ONE_SHOT will fire once (and only once) upon the first time of matching the incoming evidence against their filter.

Input nodes are additionally responsible for detecting event-relative ordering. Using two unique approaches we have incorporated event-relative temporal ordering into our specialized graphical models. These methods are discussed below in section 3.4.

3.3.2 Middle Nodes

Middle nodes are responsible for aggregating evidence from input nodes (see Appendix C). Each middle node has one or more input node parents and a conditional probability table spelling out the associated posterior probabilities with each combination of its parents' truth-values. Since a middle node is the second layer in our hierarchical model and combine various observations of input evidence, they typically represent hypothetical cognitive states of a student or a stage of task completion. For example, a sample middle node may be labeled "Student understands/does not understand concept *C*" or "Student has completed task *T*".

3.3.1 Output Nodes

Output nodes are terminal nodes and constitute the final layer of our hierarchical model (see Appendix D). The output node is partially defined by a set of actions it is to perform and its basic function is to use inference to determine whether or not to perform its associated actions. After calculating its posterior probability, instead of propagating it, the output node tests it against a defined threshold. If the posterior probability exceeds the threshold, the output node performs any associated actions it has. Otherwise, it sits dormant and waits to be triggered again.

An extensive action language defines the actions for an output node to perform given an exceeded threshold. Actions are meant to be an interactive apparatus whereby the model can externally indicate its state. The action language allows for feedback to various output destinations, such as e-mail, hard disk, and the forum. Additionally, the action language contains variables that allow a generic model-level description to eventually translates to student-specific, individuated feedback. A final feature of the action language is time dependence for actions. Actions are defined by a time element such that they are only performed during the specified time period. This will be discussed in further detail below with respect to the GMA Job System.

Using the basic constructs discussed above, we supply a simple example model that determines whether a student has performed both zoom in and zoom out actions in PixelMath (see figure 3.4). The model contains two input nodes

corresponding to the evidence we desire to recognize, a middle node to aggregate this evidence, and an output node by which we email the results.

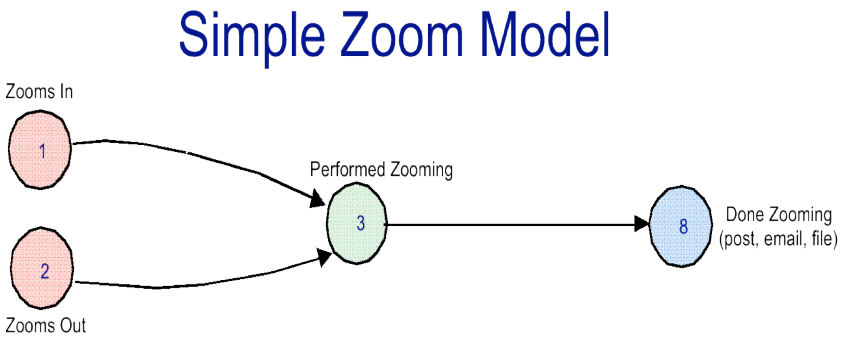

Figure 3.4
Simple PixelMath Zooming Model

3.4 Event-relative Sequence Patterns

Input nodes contain the additional feature of being able to discern relative event ordering. The ability to distinguish event-relative ordering adds another degree of power to our models since many organized educational activities involve order-dependent sequential tasks. To embed this functionality into our models we have taken two distinct approaches as outlined below.

3.4.1 A First Approach: Input Node Enabling

Suppose we wish to require that a student first perform a "zoom in" event in PixelMath prior to performing a "zoom out" event (see figure 3.5). Ideally, the input node recognizing the "zoom out" event would ignore incoming events until it realizes that a "zoom in" event has occurred.

In order to allow for selective, order-depending event listening we allow input nodes to be in one of two states: enabled or disabled. Enabled nodes function just as they did before, namely they listen to all incoming events and test them against their conditional filters. Disabled nodes ignore all incoming events.

To facilitate these two input node states we augment the traditional input node with two additional characteristics. Input nodes are now endowed with a threshold and have zero or more other input nodes as parents. Input nodes at the highest level (i.e. those without parents) are automatically enabled. Input nodes at subsequent levels (i.e. those with parents) are naturally disabled. Disabled input nodes are entirely dependent on their input node parents and must compute a probability value based on its parent's posterior probabilities. If the computed probability exceeds its defined threshold, then the input node becomes enabled and can begin listening and responding to incoming events.

This extension uses preexisting constructs to allow for simple event-relative ordering by allowing input nodes to be conditionally dependent on the truth-values of their input node parents.

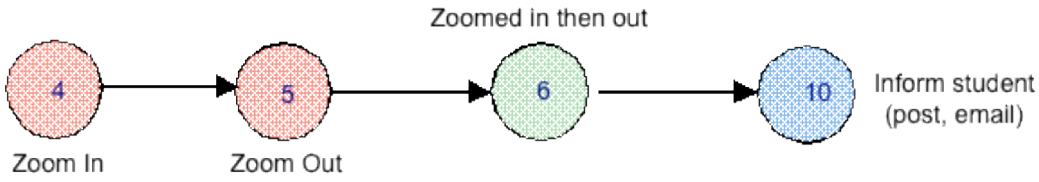

Figure 3.5
A Simplistic Event Sequence-dependent Model for Zooming in PixelMath

3.4.2 A Second Approach: Token Passing

Though input node enabling provides simple event-relative ordering constraints in our specialized graphical models, we lack the ability to model more complex sequences of events that can be expressed by finite-state automata. Consider, for example, a model that aims to predict when a student is throwing wild guesses at a question. We may choose to naïvely model this by looking for alternation between a student submission event and an "Undo" event (see figure 3.6). Using the input node enabling method we would have to create a long dependency chain of alternating input nodes – every odd node listening for submissions and every even node listening for an "Undo" event. Not only is the input node enabling solution messy in such an example, it is also impossible to model an infinite alternation between these two events. For additional expressiveness we have implemented a second event-ordering constraint solution by introducing tokens into our models.

Our token-based networks continue to allow for any input node to have zero or more parents and maintain the two states of input nodes (enabled or disabled). In addition, each input node contains the ability to hold at most one "token". A token is merely a bit set at each input node; either the input node has one, or it does not.

To support tokens, the conditional filter language for input nodes needs extension. Now, in addition to selectively choosing which evidence to listen for, input nodes can demand that one or all of its parents have a token before listening and responding to events. In this implementation, we say that input nodes waiting to be enabled by a tokenized parent contain *token dependencies*. An input node is considered disabled if it contains unmet token dependencies. If all token dependencies are met, then the input node is enabled. At this point, if the input node successfully witnesses the evidence it is listening for, it grabs the token from its parents. In other words, any time an enabled input node fires, it removes any tokens its parents may have and transfers the token to itself.

We take one final step in harnessing the power of token passing. The conditional filter language has been extended to allow for token dependencies to be coupled with event description subconditions. This feature allows for conditional statements such as "If Parent A has token AND Event E matches description X, OR if Parent B has token AND Event E matches description Y, then fire." With this final extension, input nodes can specify arbitrarily complex conditions involving any combination of conjunctions and disjunctions. Our models are now equipped to recognize arbitrarily complex patterns and sequences of events recognizable by Chomsky Type-3 grammars and finite-state automata [6].

Below we outline a model that uses token functionality (see figure 3.6).

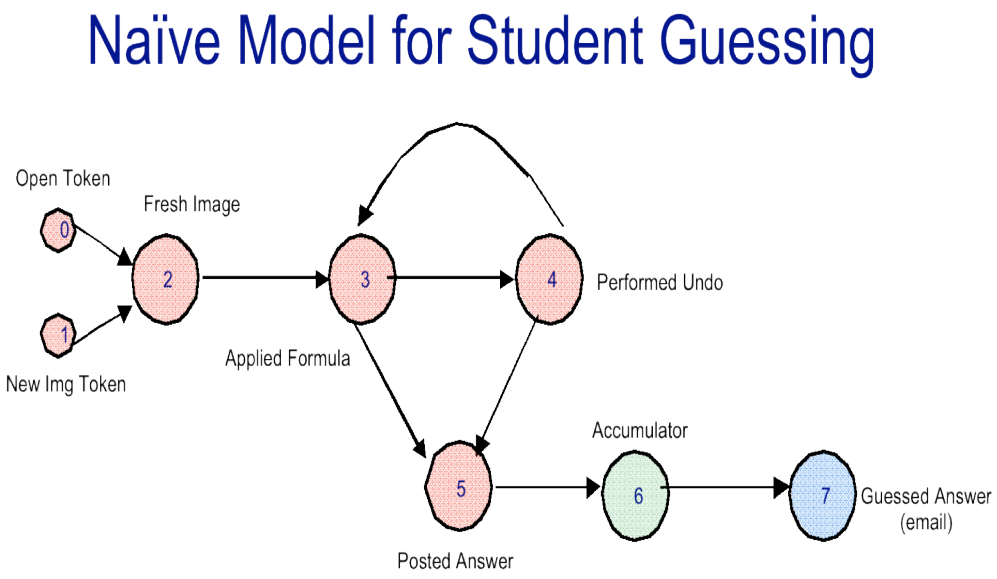

Figure 3.6
Naïve Token-based Model for Student Guessing

3.5 Model Implementation Considerations

There has been an intentional lack of implementation details located throughout the discussion of the basic constructs of our specialized graphical models. This section touches on some of the more specific implementation issues involved in creating such models. It should also be noted that Appendices A–D focus on the technical particulars of implementing one of our specialized graphical models.

GMA Models are implemented as an XML-formatted file containing descriptions of its internal components. The GMA Gnome parses the XML-based model and loads it into memory. Since we desire to provide individuated feedback, it is necessary to store student-specific data for each node in the model.

Since we need one instance of this data structure for every node in the model per student, we have created a special data structure that stores the minimum amount of unique data per student. This involves storing the current posterior probability of the node, its token state, and a counter for the number of times it has been activated and fired. It is unnecessary to store persistent data that is shared across all students such as prior probabilities, conditional probability tables, and other inter-network node characteristics.

Since we will need to frequently access nodes for various students, it is crucial that we have a fast way of accessing the data structure. For constant-time access to the data structure we store the structure in a hash table keyed on a unique string concatenation of the student ID and node ID.

4 The GMA Gnome

Now that we have set the contextual and theoretical background for the GMA Gnome, it is time to describe exactly what it is and what it does. The GMA Gnome is one of the active Gnomes running in the Gnome pool of the Gnome Environment in INFACT. It provides automated assessment to students and teachers by means of a specialized graphical model as described above.

4.1 GMA Gnome Responsibilities

The GMA Gnome is responsible for a number of tasks. First and foremost, the GMA Gnome loads one of the specialized graphical models into memory. More specifically, it creates and initializes all of the unique student-specific data structures associated with the model. It then binds itself to listen to incoming events from INFACT based on the specifications of the input nodes of the model. The GMA Gnome is responsible for feeding incoming evidence to the model, updating the model, and performing any specified actions that may result from an update of the model.

4.2 GMA Gnome Interaction Mechanisms

We have implemented a number of systems that offer us various means of tracking, controlling, and interacting with the GMA Gnome.

4.2.1 The GMA Job System

The first of the interaction systems grants us a way of scheduling the GMA Gnome. The basic runtime unit for the GMA Gnome is the GMA Job. The GMA Gnome contains two lists of associated jobs: an inactive job vector and an active job vector. The inactive job vector is a sorted list of scheduled GMA Jobs that have yet to begin. When the inactive job at the head of the list is scheduled to begin, a timer goes off and the GMA Gnome loads the job. The active job vector is a sorted list of scheduled GMA Jobs that are currently running. When the active job at the head of the list terminates, the GMA Gnome unloads the job and considers it to be complete.

GMA Jobs are the basic unit of control for assessment. The GMA Job contains four major components: a start and end time, an associated model, the name of the forum to listen for events from, and an enumeration of the particular groups and students to assess using the model.

One feature of the GMA Job System is the flexibility of the intervals specified by start and end times (see figure 4.1). The first possibility is to schedule a job to take place entirely in the past. That is, one can define a job to have a start and end date that both reside in the past. Exploiting the fact that we have a complete chronicle of student input data in the INFACT database, the GMA Gnome makes

queries to the database for the specified, past interval, builds events from the results, and replays the events to the GMA Gnome. Defining this time interval can be particularly useful in providing retrospective assessment and in testing and tweaking new models.

A second possibility is to bridge the past-future gap with a start time beginning in the past, and an end time set in the future. In this case, the GMA Gnome queries, assembles, and replays all past events while concurrently queuing all present events. When the historical playback finishes, all events that occurred during the process of building and replaying events are played back to the Gnome. At this point, the GMA Gnome considers all incoming events in real-time until the specified end time of the job. Constructing a job with this type of time interval can be helpful in establishing state in a model before performing automated assessment in real-time.

A final potential job time interval is to set both the start and end times in the future. The GMA Gnome will place a job of this specification on its inactive job queue to await loading and execution.

One ramification of job time flexibility is that it may not make much sense to perform certain specified output node actions retrospectively. For example, if we are performing retrospective assessment from the previous year it will not make much sense to email the student or to send them an instant message with automated feedback. To avoid such problems our action language contains additional constructs to specify whether or not to perform an action if the threshold is exceeded in the past.

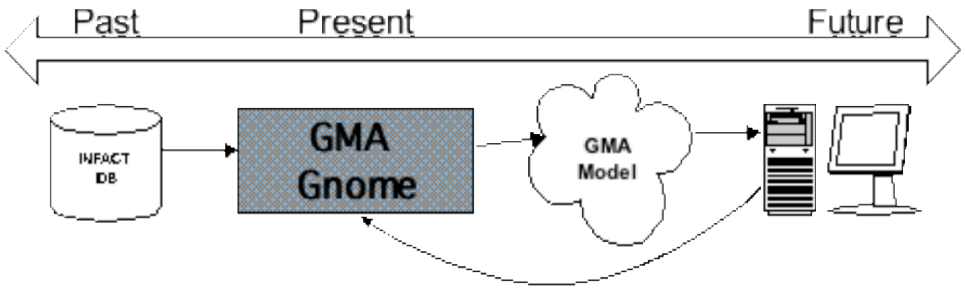

Figure 4.1
GMA Job System Time Interval Flexibility

4.2.2 GMA Status Reports

As the GMA Gnome goes about its business of providing automated assessment, we may wish to check in on its progress. GMA Status Reports are a way of displaying relevant statistics of what is happening behind the scenes of the GMA Gnome (see figure 4.2). Status reports supply us with information on how many jobs are currently loaded and their respective status, how many students are being tracked, how many events have been observed, and how many events have triggered probability updates within the models.

Message 425 in Test, group Auto

Subject: Detailed GMAGnome Report
From: Auto Notifier <nathanev@cs.washington.edu>
Date: 2004-06-28 15:36:12

```
****************************************
*  Detailed Status Report for GMAGnome  *
****************************************

--> Gnome Status <--

ACTIVE

--> General Job Overview <--

Active Jobs: 3
Inactive Jobs: 1
Number of Loaded Models: 29

--> Job Details <--
```

Job ID	Job Name	Job Status	Events Tested	Events Processed	Prob. Updates	Actions Taken
2	Run Now Job	ACTIVE	0	0	0	0
3	Single Group Job	ACTIVE	0	0	0	0
4	Group and User Job	ACTIVE	0	0	0	0
1	Sample Job	INACTIVE	0	0	0	0

Figure 4.2
Detailed GMA Status Report

4.2.3 GMA Gnome Feedback

The final interaction mechanism is directly linked to output node specifications of actions. The GMA Gnome provides feedback to students and teachers according to actions specified in its associated model's output nodes. It currently supports feedback via writing to an interactive chat-based window (IWindow), email, posting to the INFACT forum, and writing to log files on disk. As previously mentioned, the GMA Gnome supports any number and combination of output sources and its action language includes back-referenced variables that are automatically substituted to provide individualized feedback (see figure 4.3)

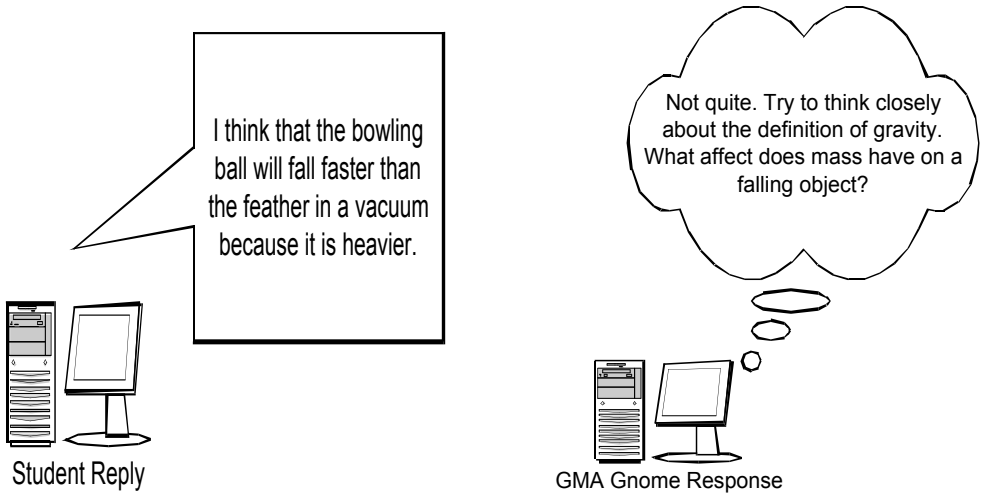

Figure 4.3
Sample GMA Gnome Automated Feedback Response

4.3 GMA Gnome Applications

The GMA Gnome can be used for numerous tasks. Below I outline a couple potential applications of the GMA Gnome given certain types of models.

• Task Completion:

The GMA Gnome is well suited to make predictions and decisions based on sequential task models. Many educational activities follow a rather rigid, linear order in which students progress. Creating a model to provide feedback at various stages of an activity is relatively trivial. One such example is the Mona Lisa Model, a model designed to indicate student progress at a sequential exercise crafted to teach students about pixels using the *Mona Lisa* and PixelMath (see figure 4.4).

• Activity Pattern Recognition:

Models can be created to capture and bring attention to common misconceptions and errors that students are making. This type of model can help provide automated feedback to the students on how to better attack the material, and may inform the teacher that some part of the activity is unclear.

• Cognitive Predictions:

A broad category of models can be created with the intention of modeling some aspect of student cognition. Since we lack perfect models of student cognition this is an intractable problem. Models of this sort are by far the most difficult yet interesting application of the GMA Gnome. The difficulty lies in creating a sufficiently complex, realistic model that can make accurate cognitive state predictions.

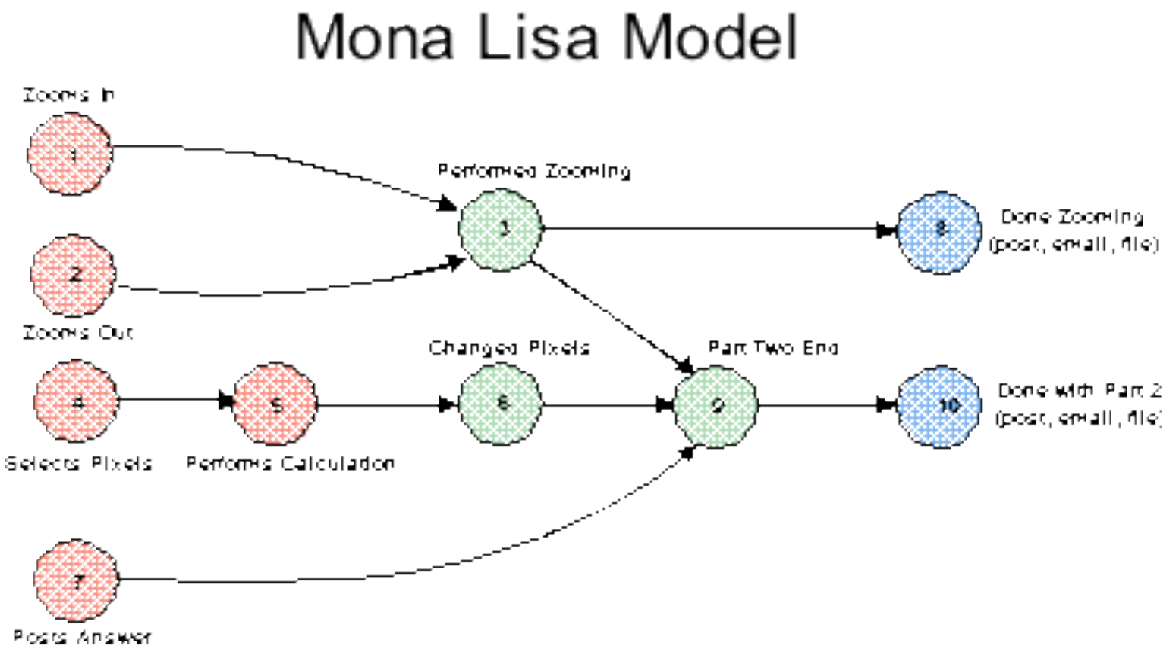

Figure 4.4
Mona Lisa Task Completion Model

5 The GMA System

In addition to providing underlying complexity and power, we want automated assessment to be easy to administer. To simplify the process of administering automated assessment with the GMA Gnome we have created a supplemental suite of software designed as an end-user interface that abstracts the more technical functionality discussed throughout this paper (see figure 5.1).

Since both GMA Jobs and Models are based on XML files that often contain intricate condition specifications, they can be cumbersome to manually create or edit. The GMA Editor and GMA Job Scheduler have been created to alleviate this burden.

Without tools written to explicitly communicate with the GMA Gnome, users must go through the laborious task of inputting textual commands to the Gnome in a designated region of the INFACT forum. The GMA Job Scheduler and GMA Job Monitor together serve to simplify the process of GMA Gnome control.

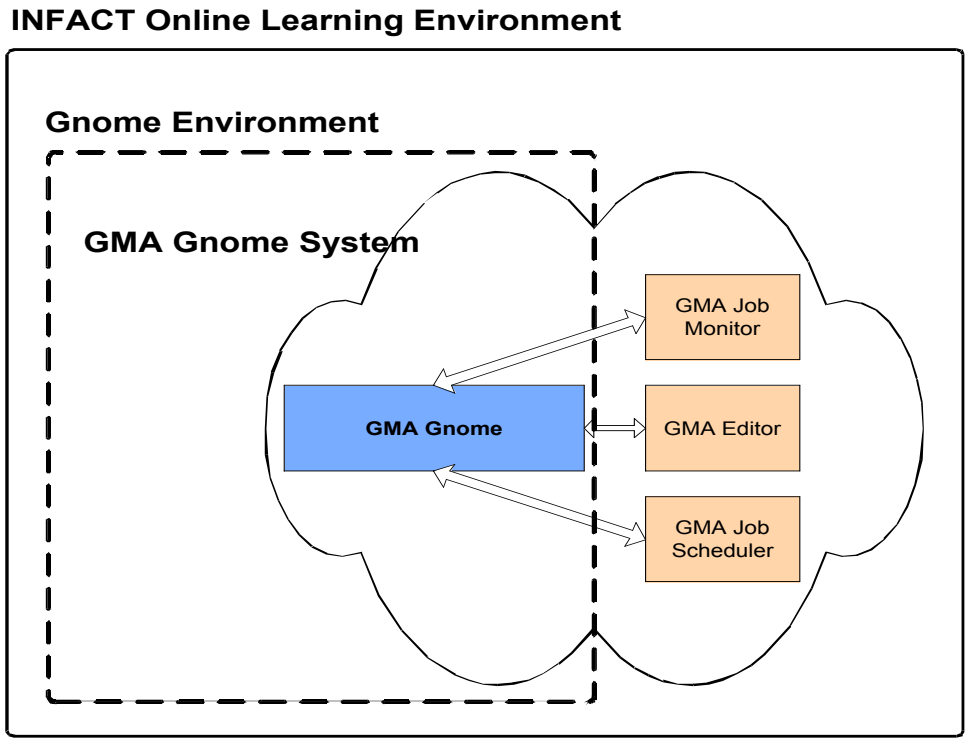

Figure 5.1
The GMA System

5.1 GMA Editor

The GMA Editor is a Java applet available through INFACT that allows for educational researchers to graphically create and edit specialized graphical models of the sort described in section 3 (see figure 5.2). Modelers have the capability to add, remove, edit, and interconnect nodes of a model. They can define the individual characteristics of each node such as prior probabilities, conditional probability tables, evidence recognition filters, and actions.

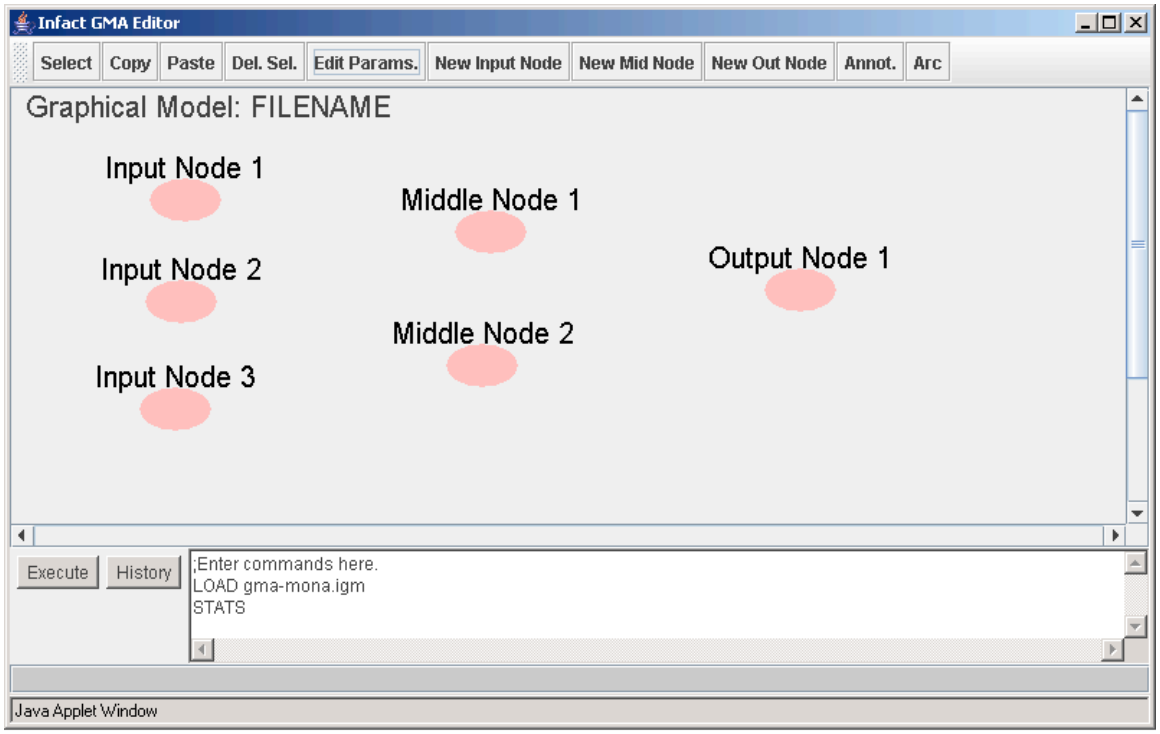

Figure 5.2
Snapshot of GMA Editor

5.2 GMA Job Scheduler

The GMA Job Scheduler is a CGI-based script written to create, edit and deploy GMA Job files. It allows users to visually define the parameters of the GMA Job including start and end dates, the associated model and forum, and which users to perform assessment on. Upon completion of job specification, designers have the ability to schedule the GMA Job thereby adding it to either the active or inactive GMA Gnome job vectors.

5.3 GMA Gnome Controller

The GMA Gnome Controller is a centralized control panel that gives end users easy access to control commands that the GMA Gnome interprets. A user may reset the state of the GMA Gnome, queue GMA Jobs, dequeue GMA Jobs, and supply arbitrary commands to the GMA Gnome. In addition, links to other GMA Gnome tools are provided for ease of access.

5.4 GMA System Workflow

With the suite of software available in the GMA System there is a clear step-by-step process whereby one can start from scratch and end up providing automated assessment (see figure 5.3).

One begins by creating or editing a GMA Model with the GMA Editor. The model will specify how the GMA Gnome will act given certain input within INFACT. Next, one must create a GMA Job file that uses this model. At this point it is time to use the GMA Job Scheduler to schedule the GMA Job and have the GMA Gnome start listening to and processing events (or queue the job if it is to start in the future). Once the job is loaded and acting, we can observe how the model is reacting to student input using the GMA Gnome Controller. If the model is not capturing exactly what we want, we can go back to the first step and tweak the model to work as we wish.

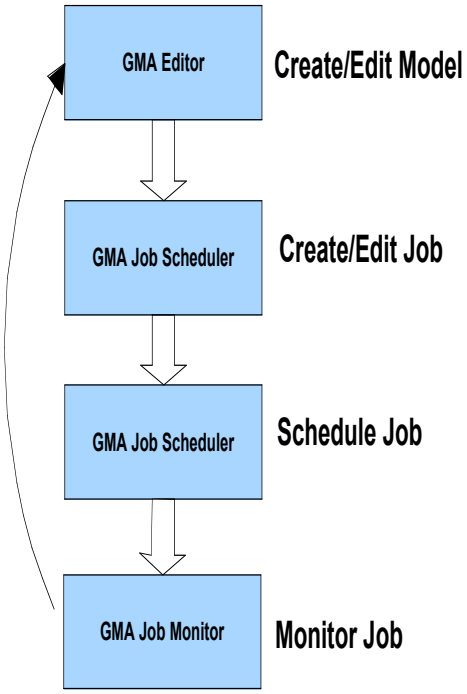

Figure 5.3
GMA System Workflow Diagram

6 Conclusions

I wish to conclude by examining one GMA Gnome experiment and enumerating future work and research that can be done with the GMA System.

6.1 Preliminary Findings: The GMA Gnome In Action

The GMA Gnome was put to the test in the summer of 2004 on a group of 19 high school MITE students learning about pixels via PixelMath [6]. The seminar involved working through an activity worksheet designed to introduce students to the basics of pixels and PixelMath by performing various manipulations on a digital copy of the *Mona Lisa*. Since the activity worksheet created to follow the instructions in sequential order, we created a GMA Model called "The Mona Lisa Model" to model student progress at various stages of the worksheet (see figure 6.1).

The worksheet asked students to first zoom in and out on the *Mona Lisa*. It then asked them to select a group of pixels and perform some PixelMath calculation upon them. Finally, they were to post to the INFACT forum their results. A different region of the Mona Lisa Model captures each of these worksheet features.

The GMA Gnome successfully served as a task completion monitor by accurately predicting and reporting completion of certain activity tasks of individual students in real-time. To test various output sources, the Mona Lisa Model was created to generate a forum post, write to a file, and email the seminar administrators with the results of each student's progress.

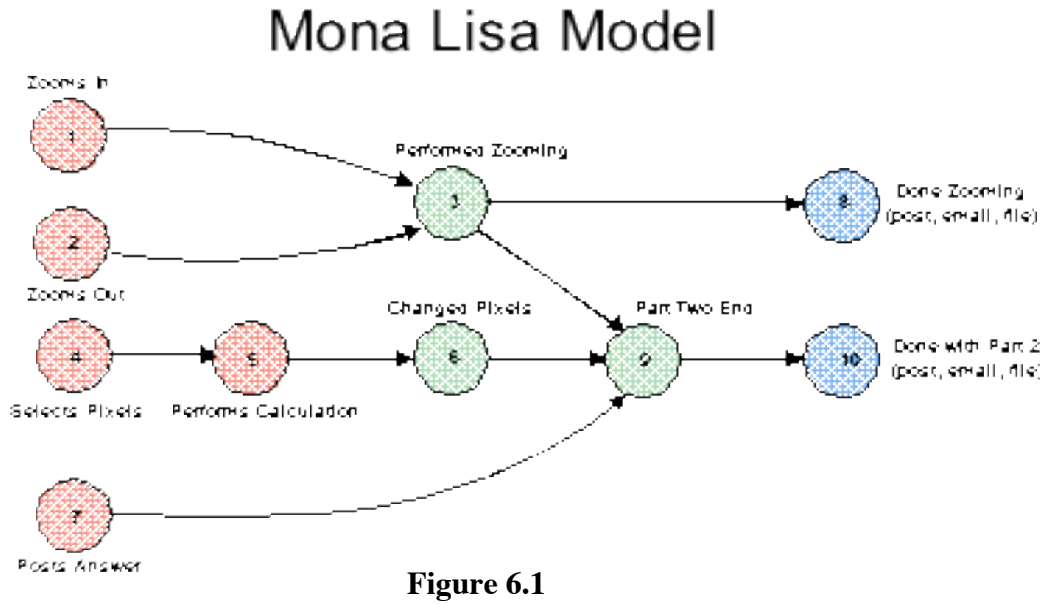

Figure 6.1
The Mona Lisa Task Completion Model

6.2 Future Work

There is much room for future work in the GMA System. Below I highlight numerous ideas of places we can go now that the groundwork has been laid.

- Additional Pattern Recognition:

  In order to obtain a truly comprehensive picture of a student's understanding, we desire to fuse together as many types of data as we can. Since online learning environments are rich in the types of input they allow (e.g. textual input, graphical input, audio input, and input from other devices), having a way to translate these alternative forms of input into evidence that can be fed to our models is essential. As it stands, the GMA System supports two primary forms of basic pattern recognition: textual and UI-based. Future work could include expanding this set of pattern recognition using computer vision techniques, semantic natural language processing, and voice recognition.

- Real-time Feedback:

  Ideally, INFACT would have an interactive chat client where the GMA Gnome could automatically post feedback to students in real-time. This would help in making the automated assessment as unobtrusive as possible. Note: as of Summer 2005 this is done!

- Job Monitor Enhancements:

  Instead of simply providing basic job statistics, the job monitor could be extended to provide student-specific job statistics. There could be a program that displays a graphical representation of the model being used by the job and individual student progress in that model.

- GMA Model work:

  Much of the future work to be done with the GMA Gnome is in creating a solid base of models to use. Creating sufficiently complex, accurate models will help test the GMA Gnome and its capabilities. In general, the GMA Gnome needs to be further tested on more models and in more classroom experimentation. One interesting potential for future work is to look into machine learning techniques that may aid in automatically generating relevant models. This would alleviate the burden of model creation currently placed on the domain expert.

6.3 Acknowledgements

The author would like to personally thank Steve Tanimoto, Adam Carlson, and Nick Benson for dedicating help in providing ideas, support, and time toward the GMA Gnome project.

Bibliography

[1] Minstrell, J. 1992. Facets of students' knowledge and relevant instruction. In Duit, R., Goldberg, F., and Niedderer, H. (eds.), Research in Physics Learning: Theoretical Issues and Empirical Studies. Kiel, Germany: Kiel University, Institute for Science Education.

[2] Tanimoto, S. L. 1995. Exploring mathematics with image processing. Proceedings of the 1995 IFIP World Conference on Computers in Education, Birmingham, UK, July 23-28. London: Chapman & Hall, pp.805-814.

[3] Benson, Nicholas J. 2004. it The INFACT Gnome Environment: A Platform for Autonomous Agents Generating Automatic Student Assessments and Feedback. Senior honors thesis, Dept. of Computer Science and Engineering, University of Washington, Seattle. http://www.cs.washington.edu/ole/benson-thesis.pdf.

[4] Norvig, P. Russel, S. 2002. Artificial Intelligence: A Modern Approach (2nd Edition). Prentice Hall.

[5] http://www.wikipedia.org/Graphical_model. As accessed May 31, 2005.

[6] Tanimoto, S. Evans, N. Carlson, A. 2006. Sequential Input Graphical Model Assessment Diagrams for Probabilistic Analysis of Event Streams. Submitted.

Appendix A: GMA Models

Outline

This appendix will outline the basic mechanisms for creating GMA Models. GMA Models are the basis of providing assessment with the GMA Gnome. There are two general methods used to create GMA Models: the GMA Editor and manual editing of a ".igm" XML file. The appendices highlight all of the syntax and key features embodied in creating models.

Composition

Models are constructed of three sets of nodes, each covered in detail in their own appendix:

- Input Nodes (Appendix B)
- Middle Nodes (Appendix C)
- Output Nodes (Appendix D)

XML Descriptions

Though its individual parts define the majority of a GMA Model, there are two model-level XML descriptions worth noting. All GMA Models begin and terminate with the following XML tags:

```
<model> ... Model Description ... </model>
```

The second model-level construct is in defining the model name. To define the model name, place the model name string between the following tags:

```
<modelname> Example Model </modelname>
```

Appendix B: Input Nodes

Outline

Input nodes are the most intricate of the nodes in a model. They handle evidence recognition with a conditional language and can observe event-relative ordering via input node enabling or token passing.

Composition

Input nodes contain the following fields:

{Basic Input Nodes}
- A unique node ID
- A prior probability
- A particular event type to listen for
- A selective filter
- A response method

{Input node enabling}
- A set of input node parents (optional)
- A parent-associated conditional probability table (optional)
- A threshold (optional)

{Token passing}
- A token state (optional)

{GMA Editor Displaying}
- A node label (optional)
- X and Y coordinates (optional)

Basic Input Nodes

Writing the most basic input node requires inclusion of the first five properties above. We begin each input node definition with the following XML tags:

<inputnode> ... Input Node Description ... </inputnode>

Within the input node description, each input node in a GMA Model must contain a model-wide, unique node ID:

<nodeid>X</nodeid>
(where X is some integer value)

Each input node must also contain a prior probability. This value indicates a description of what is known in the absence of evidence. A prior probability is established as follows:

<prior>Y</prior>
(where Y is some double from 0.0 to 1.0)

The next necessary attribute of an input node is an event type to listen to. Event types are embedded within events dispatched to the Gnome. If the event type of an incoming event does not match the specified event type, the input node will ignore the incoming event. Example event types include: 'pixelmath', 'forumpost', and 'tokenholder'.

<eventtype>type</eventtype>

A basic input node must also specify which events it is interested in by means of the conditional language. Incoming evidence that matches the "filter" is considered evidence and may update the model. Evidence that does not match the filter is ignored. For now we'll gloss over the conditional language specifics and simply show what tags to wrap around it. Later in this appendix you can find a full description of the conditional language.

<filter>{*filter1*}{*filter2*}</filter>
(where filter1 and filter2 are individual filters as described below)

The final ingredient to a basic input node is its response method. This determines how it responds to evidence that matches its filters. The specifics for response methods are included in a separate section below.

<responsemethod>(*method1*)(*method2*)</responsemethod>

Conditional Language

The input node conditional language allows input nodes to precisely specify which events to consider as evidence. Filter constructs are generally dependent on the type of event we seek. For example, if we are looking at 'forumpost' events, we expect there to be an associated subject and body that we can match against. If we are looking at 'pixelmath' events, we expect there to be a string describing which UI event was submitted. Illustrating how to create evidence filters for these two types of events will be instructive in showing how to extend the system to allow for other sources.

An input node can combine any number of filters and multiple filters are taken as a conjunction. That is, as soon as one filter fails to match, the remainder filters are not assessed and the evidence is ignored. Filter conditions are clustered in curly braces:

<filter>{CONDITION 1}{CONDITION 2}</filter>

It should also be noted that we consider a regular expression to "match" if it merely contains the regular expression input into the filter. That is, the regular expression below will "match" the following sentence:

Hello, my name is Bob.
/Hello/

• Pixelmath Event Filtering

Events coming from PixelMath come in the form of a descriptive string indicating which UI event generated the event and any particular data associated with the UI event. For example, clicking on the “Undo” button in PixelMath generates the event string:

```
{Description,WindowNum=3;MenuItem=java.awt.Event[id=1001,x=0,y=0,target=java.awt.
MenuItem[menuitem2,label=Undo],arg=Undo}
```

To match such an event description string the input node’s filter needs to use the DESC argument followed by a regular expression aimed at matching the string. DESC tells the GMA Gnome that the UI description string needs to match the supplied regular expression in order to pass as evidence. The syntax, then, is:

```
<filter>{DESC:/regularexp/}</filter>
```

In addition to considering evidence on the basis of the the application description event string we can look at the application event type. Each event in INFACT software (such as PixelMath) also supplies a software-dependent event type on which we can filter using the TYPE argument. TYPE tells the GMA Gnome that the UI event type needs to partially match the supplied regular expression in order to pass as evidence:

```
<filter>{TYPE:/regularexp/}</filter>
```

To show a combined example, if we want to find PixelMath Undo events that are of type ‘ImageFrameWindow’, we may use the following filter:

```
<filter>{TYPE:/ImageFrameWindow/}{DESC:/label=Undo/}</filter>
```

• Forumpost Event Filtering

We expect events coming from the forum to have an associated subject and body. Accordingly, the first filter argument for ‘forumpost’ events is SUBJECT, specifying a regular expression to match against the subject of the post:

```
<filter>{SUBJECT:/regularexp/}</filter>
```

Analogously, the second filter argument for forum posts is BODY, specifying a regular expression to match against the body of the post:

```
<filter>{BODY:/regularexp/}</filter>
```

For example, if we wanted to match a post that has a subject “Hello class” and a body of “This is my post”, we may construct a general filter as such:

```
<filter>{SUBJECT:/Hello\w+class/}{BODY:/\s+post/}</filter>
```

Again, it should be noted that for specificity it is permissible to include as many SUBJECT and BODY pattern matchers as desired within one filter.

• Extended Regular Expressions: Ranges

The final construct in the conditional language is a shortcut to perform numerical range tests embedded in regular expressions. Any condition language argument (be it TYPE, DESC, SUBJECT, or BODY) can precede its regular expression with the character "r" to indicate that it contains a range. This indicates to the GMA Gnome that a back-referencable range is embedded within the regular expression and it ought to check the range bounds supplied at the end of the regular expression. For example, the following filter condition indicates that there are two ranges in the regular expression. The first number must be some number between 40-100 and the second must be from 1000-2000:

```
{SUBJECT:r/reality (\d+) or (\d+)/,40-100,1000-2000}
```

Response Methods

As described in section 3.3.1, input nodes have three ways of responding to evidence: REQUIRE, ACCUM_ZENO, and ONE_SHOT. Response methods are placed between the <responsemethod> tags (see above) as with the following syntax:

REQUIRE:	(REQUIRE *N*)
ACCUM_ZENO:	(ACCUM_ZENO *X*)
ONE_SHOT:	(ONE_SHOT)

It sometimes makes sense to make the response method a combination of ONE_SHOT and REQUIRE. For example, if we wanted an input node to fire only once after observing five events that matched its filter, we may define the input node as follows:

```
<responsemethod>(ONE_SHOT)(REQUIRE 5)</responsemethod>
```

"Enabling" Input Nodes

Input node enabling is the first of the two methods of providing event-relative temporal ordering (see section 3.4.1). To use it, input nodes now have parents, thresholds, and conditional probability tables. As such, these three constructs are built into the input node description. To specify the parents of an input node, we use a comma-delineated list of parent node IDs:

```
<parents>nodeid1,nodeid2</parents>
```

(where nodeid1 and nodeid2 are the nodeids of the parents)

We now define a conditional probability table based on the number of parents the input node has. It will use this conditional probability table to calculate its posterior and determine if it is enabled or not. The conditional probability table is a comma-delineated list of probabilities corresponding to the truth-values of every combination of its parents. The list begins with all parent's truth-values (i.e. whether they have fired or not) as false and counts up in binary to all parent's truth-values being true. So, if we had two parents, the probability distribution may abstractly look like this:

```
<pd>FF,FT,TF,TT</pd>
```

(where each truth scenario is a double from 0.0 to 1.0)

It should be noted that there will be $2^{\text{number of parents}}$ entries in each probability distribution. Finally, an input node can now have a threshold that it uses to gauge whether or not its posterior probability indicates it is "enabled" given the truth-values of its parents. The threshold is defined as follows:

<threshold>X</threshold>
(where X is some double from 0.0 to 1.0)

Token Passing Input Nodes

The second method of noticing event sequence is by using tokens as described by section 3.4.2. The main construct for declaring a token dependency on one of an input node's parents is to use the TOKENDEP argument. Dependencies on different parents are separated by parentheses in this general form:

{TOKENDEP:*(dep1)(dep2)*}

Unlike action conditions, individual token dependencies are considered as a disjunction. That is, if any of the dependencies are true, the token dependency will be matched. Essentially token dependencies are considered in the form "If any of these parents have a token, then consider the evidence." If the modeler wants the input node to be dependent on all of its parents, he or she can take a shortcut and simply enter:

{TOKENDEP:-1}

For full expressivity each individual dependency may contain its own action conditions. Just as with standard action condition filters, actions that are embedded into the disjunctive token dependencies are conjunctive. This allows for conditions in the form "If parent X has a token AND the incoming event E matches D, OR parent Y has a token AND the incoming event E matches F, then fire." Individual dependencies, as mentioned above, are separated by parentheses. The first character following the opening parentheses must be the node ID of the parent the token is required from. Following this node ID are the associated action conditions. For example, requiring a token from parent with node ID 23 and requiring that the incoming event match a simple body regular expression may be expressed like this:

{TOKENDEP:(23:{BODY:/regexp/})}

A further example is in requiring a token from one of two parents:

{TOKENDEP:(23:{BODY:/regexp/})(24:{SUBJECT:/regexp/})}

It is easy to see that token dependencies allow for any arbitrary combination of ANDed and ORed statements and are fully expressible. It is also easy to see that the syntax can become rather convoluted. This warrants using the GMA Editor.

Graphically Displayed Input Nodes (GMA Editor)

The final set of parameters in describing an input node is used by the GMA Editor to graphically display the input nodes. These involve the X and Y coordinates to display the input node at in the software and what label to attach to the input node. The X and Y coordinates are defined as follows:

<x>xcoord</x>
<y>ycoord</y>

(where xcoord and ycoord are integer values of the pixel location)

Lastly, we define an input node's label:

<label>Input Node Label</label>

```
<inputnode>
<nodeid>1</nodeid>
<label>Zooms in</label>
<prior>0.1</prior>
<eventtype>pixelmath</eventtype>
<filter>{DESC:/ZoomIn|Zoom In/}</filter>
<responsemethod>(ACCUM_ZENO 0.7)</responsemet
</inputnode>
```

Figure I
Example XML-defined Input Node

Appendix C: Middle Nodes

Outline

Middle nodes are relatively simplistic. They are defined only by a few constructs, all of which are also used in defining input nodes. See Appendix B for further description of these constructs.

Composition

Middle nodes contain the following fields:

- A unique node ID
- A prior probability
- A set of parents
- A parent-associated conditional probability table

{GMA Editor Displaying}

- A node label (optional)
- X and Y coordinates (optional)

Basic Middle Nodes

All four of the necessary criteria for defining a basic middle node involve ideas already discussed in Appendix B. We begin by defining a middle node region:

<midnode> ... Middle Node Description ... </midnode>

We must give our middle node a unique node ID:

<nodeid>X</nodeid>
(where X is some integer value)

Additionally, we define a prior probability:

<prior>Y</prior>
(where Y is some double from 0.0 to 1.0)

Next, we specify the node ids of any of its parents. Note that middle nodes can have input nodes and middle nodes as parents.

<parents>nodeid1,nodeid2</parents>
(where nodeid1 and nodeid2 are the nodeids of the parents)

Finally, we give the middle node a conditional probability table that relates different truth conditions of its parents to a probability.

<pd>FF,FT,TF,TT</pd>
(where each truth scenario is a double from 0.0 to 1.0)

Graphically Displayed Middle Nodes (GMA Editor)

The final set of parameters in defining a middle node is for graphical display by the GMA Editor. The X and Y coordinates for inter-software middle node location are defined as follows:

```
<x>xcoord</x>
<y>ycoord</y>
```

(where xcoord and ycoord are integer values of the pixel location)

Lastly, we define a middle node's label:

```
<label>Middle Node Label</label>
```

```
<midnode>
<nodeid>3</nodeid>
<label>Completes zooming</label>
<prior>0.1</prior>
<parents>1,2</parents>
<pd>0.1,0.3,0.3,0.8</pd>
</midnode>
```

Figure II
Example XML-defined Middle Node

Appendix D: Output Nodes

Outline

Output nodes are similar to input nodes that use input node enabling. Their main responsibility is to define actions that are performed if their threshold is exceeded upon performing a probability calculation. See Appendix B for further explanation of some of the basic output node constructs that are identical to input node constructs.

Composition

Output nodes contain the following fields:

- A unique node ID
- A prior probability
- A set of parents
- A parent-associated conditional probability table
- A threshold
- Associated actions

{GMA Editor Displaying}

- A node label (optional)
- X and Y coordinates (optional)

Basic Output Nodes

The first four of the necessary criteria for defining a basic output node involve ideas already discussed in Appendix B. We begin by defining an output node region:

<outnode> ... Output Node Description ... </outnode>

We must give our output node a unique node ID:

<nodeid>X</nodeid>
(where X is some integer value)

Additionally, we define a prior probability:

<prior>Y</prior>
(where Y is some double from 0.0 to 1.0)

Next, we specify the node ids of any of its parents. Note that output nodes can have input nodes or middle nodes as parents.

<parents>nodeid1,nodeid2</parents>
(where nodeid1 and nodeid2 are the nodeids of the parents)

Now we give the output node a conditional probability table that relates different truth conditions of its parents to a probability.

<pd>FF,FT,TF,TT</pd>

(where each truth scenario is a double from 0.0 to 1.0)

One of the key characteristics of an output node is its threshold. If the threshold is exceeded after performing its update() function given the values of its parents and the conditional probability table, then the associated actions are performed. A threshold is defined as such:

<threshold>X</threshold>
(where X is some double from 0.0 to 1.0)

Output nodes are primarily designed to perform actions. These actions are ways for the model to provide feedback of its state. Modelers can create models with meaningful output node actions in order to provide useful feedback. Actions are generally defined as such, where each individual action is described in the action language (see below):

<action>{*action1*}{*action2*}</action>
(where action1 and action2 are actions as defined below)

Action Language

The action language allows for an output node to communicate externally to any number of predefined external sources. At the time of writing this document, output nodes have the ability to send emails, write to log files on disk, write to an interactive chat window (IWindow) and post to the INFACT forum. As discussed in section 4.2.1, the GMA Job System allows for jobs to be contained partially in the past. Performing retrospective jobs may have implications on what actions we wish to perform. The action language contains four mechanisms that define whether to perform the action during a certain time or not.

• Standard Actions
Basic actions use the NOTIFY argument and are of the following abstract form:

{NOTIFY "*subject*" "*body*" (*source1*,*source2*,*source3*)}

NOTIFY tells the GMA Gnome to perform some type of notification action. It uses the defined subject and body of the message and writes it to each of the comma-delineated sources. The terms 'subject' and 'body' are abstract definitions that have slightly different meanings given the output source. When sending email and posting to the forum, 'subject' and 'body' correspond to the intuitive idea of an email or forum post subject and body. When writing to a file, both the subject and body are logged.

Output sources need be input as either an integer number corresponding to the group id to post to, an email address, or the location of a file on disk.

• Back-Referenced Variables

The "subject" and "body" fields of a NOTIFY action have the additional benefit of including back-referenced variables to create individual feedback. If we want to provide specialized feedback to a student we can write a general subject or body with embedded variables that the GMA Gnome translates. Each back-referenced variable begins and ends with a '$'. For example, if we want to provide a general action body that tells the time that the output node was triggered and the name of the student whose model triggered it, we specify the following body:

"Student: $STUDENTLASTNAME$, $STUDENTFIRSTNAME$ triggered output at $TIMEDONE$"

The output sources may contain back-referenced variables as well. For example, we can post to the auto forum or send an email to the student whose model generated the output.

($AUTOGROUP$, $STUDENTMAIL$)

Taken together, we define the action:

<action>{NOTIFY "Subject" "Student: $STUDENTLASTNAME$, $STUDENTFIRSTNAME$ triggered output at $TIMEDONE$" ($AUTOGROUP$, $STUDENTMAIL$)}</action>

Below we list all currently available back-referenced variables:

Variable Name	Description
STUDENTFIRSTNAME	The first name of the student
STUDENTLASTNAME	The last name of the student
FORUMINTERNALNAME	The name of the forum this job is associated with
STUDENTMAIL	The email address of the student
STUDENTID	The student ID of the student
POSTERIOR	The posterior of the Output Node
TIMEDONE	The time the action was performed
STUDENTGROUP	The group ID that the student belongs to
AUTOGROUP	The auto group ID of the associated forum

Figure IV

Output Language Back-Referenced Variables

• Time-dependent Actions

In addition to the basic NOTIFY action there are three other action types that indicate whether to email, post, or write to disk given the time that the output node is triggered. If we are performing a historical assessment, the output node will be triggered "in the past." The following table outlines the different action types and what effect they have given a present-time trigger or past-trigger.

Action Type	Triggered in Past	Triggered Presently
NOTIFY	Post, Write File	Post, Write File, Email
NOTIFY-PRESENT	---	Post, Write File, Email
NOTIFY-PAST	Post, Write File, Email	---
NOTIFY-ALWAYS	Post, Write File, Email	Post, Write File, Email

Figure V

Output Language Time-dependent Actions

Graphically Displayed Output Nodes (GMA Editor)

The final set of parameters in defining an output node is for graphical display by the GMA Editor. The X and Y coordinates for inter-software output node location are defined as follows:

<x>xcoord</x>
<y>ycoord</y>

(where xcoord and ycoord are integer values of the pixel location)

Lastly, we define an output node's label:

<label>Output Node Label</label>

Figure VI

```
<outnode>
<nodeid>8</nodeid>
<label>Zoomed!</label>
<prior>0.0</prior>
<threshold>0.7</threshold>
<parents>3</parents>
<pd>0.0,1.0</pd>
<action>
{NOTIFY "student zoomed""Student # $STUDENTID$ : Completed
zooming with certainty of $POSTERIOR$ (/usr/share/zoom.gmo,
$STUDENTGROUP$, nathanev@cs.washington.edu)}
</action>
</outnode>
```

Example XML-defined Output Node