Building Applications Using Only Demonstration

Richard G. McDaniel and Brad A. Myers
HCI Institute, School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, USA
Tel: 1-412-268-3066
E-mail: { richm, bam }@cs.cmu.edu

ABSTRACT

By combining the strengths of multiple interaction techniques and inferencing algorithms, Gamut can infer behaviors from examples that previously required a developer to annotate or otherwise modify code by hand. Gamut is a programming-by-demonstration (PBD) tool for building whole applications. It revises code automatically when new examples are demonstrated using a recursive procedure that efficiently scans for the differences between a new example and the original behavior. Differences that cannot be resolved by generating a suitable description are handled by another AI algorithm, *decision tree learning*, providing a significantly greater ability to infer complex relationships. Gamut's interaction techniques facilitate demonstrating many examples quickly and allow the user to give the system *hints* that show relationships that would be too time consuming to discover by search alone. Altogether, the concepts combined in Gamut will allow nonprogrammers to build software they never could before.

Keywords

End-User Programming, User Interface Software, Application Builders, Programming-by-Demonstration, Programming-by-Example, Inductive Learning, Gamut

INTRODUCTION

Much of the intellectual effort that goes into producing a game or an educational tool is not in programming the game's logic but in providing the engaging background, artwork, and gameplay that keeps players interested. Artists and educators that have the talent and ideas to produce good material are often unable to program computers. Therefore, providing tools which eliminate the burden of programming while still maintaining the ability to build interesting software is desirable.

Traditional methods for producing interactive software require extensive programming knowledge. Programming graphics in common environments like Visual C++ or Motif require great effort even from seasoned programmers. Tools like interface builders facilitate the design of the layout and the look of some parts of the program, but still require programming to make the interface actually work. Authoring tools like HyperCard [5] and Director [7] provide more support, but require the developer to learn baroque programming languages to produce anything beyond simple interactions. Application builders such as Klik & Play [6] which do eliminate programming also impose severe limits on the kinds of programs one can make.

A typical solution has been to simplify the language with which one programs. KidSim [15] (now called Cocoa) uses a notably simple language with pictures that show how objects change from one state to another. However, when the language is simplified this much, the tool often loses generality: KidSim requires an application to be constructed from the same rectangular tiles from which the language's pictures are made. Furthermore, even in KidSim, the developer must learn and use a separate annotation language to form conditional phrases required by all but the most basic applications.

We have applied programming-by-demonstration (PBD) techniques that change the process of programming altogether. In PBD, the user shows the system what to do by giving it examples of the desired behavior. By analyzing the examples and generalizing them using inductive learning techniques such as heuristic search and decision tree learning, the system can create code that performs the same behavior. We seek to improve PBD techniques in order to build a broader class of applications.

Our research shows that many behaviors requiring code annotation or that are not possible in other PBD systems can be generated entirely by demonstration. In fact, we have made complete applications such as Tic-Tac-Toe, a Turing machine simulation, and several maze games entirely with demonstration. Soon, we will be able to make even more advanced games like Chess, Reader Rabbit, and PacMan.

What Gamut Can Do

Gamut is designed to build applications like video games, simulations, and educational software. We are focusing on games which resemble board games that possess two-dimensional graphics and behaviors that involve how "pieces" interact with one another and interact with a background "board." By our definition, PacMan is a "board game" with a background maze with dots and pieces for the PacMan and monsters. Monsters, bullets, and other moving pieces use timers to control their actions. In this paper, we will call graphical pieces and items in the background "objects" in the standard object-oriented sense. We will also introduce other objects later to represent non-graphical concepts.

Published in Proceedings of IUI'98, the International Conference on Intellegent User Interfaces, pp. 109-116

Gamut can infer complex, conditional relationships between objects without requiring the developer to write any code. Like other PBD systems, Gamut can infer behaviors that create, modify, and destroy objects as one does in a graphical editor. But Gamut infers relationships that others cannot:

- Objects involved in a behavior can be independent from the objects modified by the behavior. In other words, the effect of a behavior can be independent from the reason the behavior occurred. For instance, in PacMan, the monsters turn blue when the PacMan eats a big dot. Other PBD systems require the developer to create such relationships by editing a code-like description of the inferred rules.
- An object's description can form an arbitrarily long chain of relationships. For example, Gamut can infer a legal move in a Monopoly game: "the square which is the dice's number of squares away from the initial position of the current player's piece." This one description combines objects such as the dice, the player turn indicator, as well as the board configuration, into one expression chain.

To make Gamut easier to use, we have assembled interaction techniques that simplify and streamline the demonstration process. First, the developer uses *hints* to point out important objects that guide the construction of new relationships. Second, Gamut uses efficient interaction techniques called *nudges* to make examples easier to generate and more numerous. These nudges can produce negative examples as easily as positive examples.

Gamut uses a recursive searching technique to scan a behavior and find how it differs from a newly demonstrated example. The algorithm combines two forms of learning to generate new expressions: *heuristic search* and *decision trees*. Heuristic search uses rules to examine the context of a demonstration. It can learn complicated expressions quickly by recognizing common situations. A decision tree is a statistical learning algorithm that can learn arbitrary boolean expressions using examples. Gamut uses decision tree learning to form the predicates of conditional statements.

What Gamut Cannot Do

Gamut's most serious flaw is a lack of feedback which is discussed in future work. Since Gamut cannot infer when the developer is making a mistake, the developer may be unaware when there is a problem. If the developer discovers a problem, though, Gamut's interaction techniques make repairing the bug quite easy.

Gamut also has domain restrictions. For instance, presently Gamut's graphics editor cannot rotate objects. A developer wanting to construct the game Asteroids, which involves a small space ship that rotates, cannot use Gamut because it does not support a necessary operation. Making Gamut support a larger domain is possible by adding more features.

Gamut also does not infer application state. Gamut infers relationships between state objects, but cannot generate new state objects by itself. So, if a behavior has to repeat five times, part of the state the user must provide is the counter which says how many times the behavior has left to go. Though some application state can be derived from existing objects (like the position of squares on a Chess board which represent where a piece is allowed move), the developer must often create extra objects to represent counters, switches, and other state variables.

Finally, Gamut can only create algorithms that the developer understands. For instance, many computer games like computerized Chess use sophisticated algorithms to model strategies in order to make a good move. Demonstrating such a strategy requires more state variables and demonstrations than any developer will be able to perform. Any algorithm beyond the developer's understanding cannot be magically inferred by Gamut. The Gamut version of Chess will not have a computer opponent, but it will keep human players from making illegal moves because the rules to Chess use state that is mostly available from the board and can be demonstrated in a reasonable number of examples.

EXAMPLE

We will use a simple example to show how to make a Gamut application. The object of the game is to steer a droplet of water into a bucket by selecting which color pipe the droplet will follow. Figure 1 shows the main Gamut window in which is drawn the game board. The background consists of a pyramid of pipes. Some are painted green and the others are painted red. At the bottom of the pyramid is a bucket. To switch colors, the player clicks the mouse button.

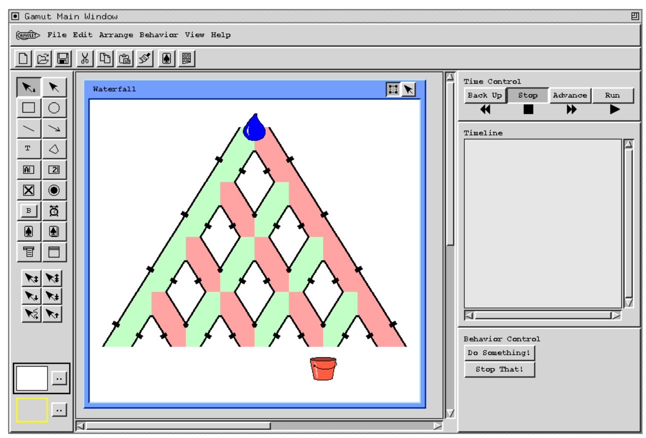

Figure 1: The main drawing window showing the final game.

This game is certainly simple, and variants of it can be created with other tools. But because the game has an element of user input (clicking to steer), and because the direction the drop chooses depends on state set by that user input, there is no prior tool that can build this game without requiring the developer to write code or annotate rules by hand.

The first step is to draw the parts of the game. Like most application builders, Gamut's primary interface consists of a graphical drawing editor/interface builder. The developer chooses graphical primitives and widgets from the drawing palette and lays them out in the work area. A simulated window pane (titled "Waterfall" in Figure 1) divides the working area into portions visible to the player and an offscreen area where the developer can put objects which store state. (This game needs no offscreen objects.) The developer draws images for the background pipes, the water droplet, and the catch bucket.

Guide Objects

In the next step, the developer draws objects into the scene that control state, but are not part of the visible elements of the game. In Gamut, we call these *guide objects* because they are meant to "guide" the actions of other objects. Guide objects are graphical objects that are visible only when the application is being built and are made invisible when the game is played. Using guide objects is not new. Metamouse [8] and Demo II [1] use a nearly identical abstraction, and Rehearsal World [3] developed "offstage actors" as its term for offscreen objects.

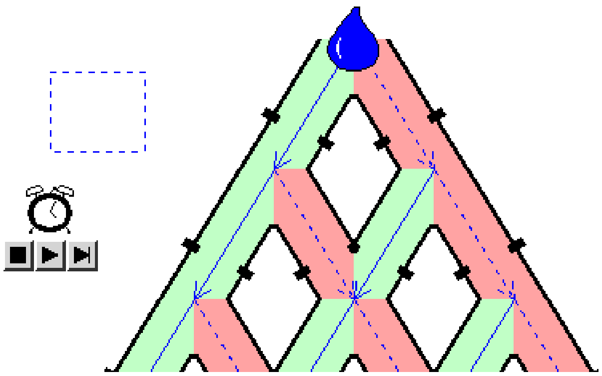

Figure 2: Guide objects for the water droplet game.

Next to the pyramid, the developer draws a rectangle to act as a state variable to show which path the droplet follows and adds a timer to make the droplet move independently. Since the pipe images are bitmaps, and the computer cannot interpret the image inside them, the developer draws arrow lines on the diagram one along each pathway. The developer uses a dotted line to show moving right and a solid line to move left. The completed picture is shown in Figure 2.

Demonstrating Behavior

With all the graphics drawn, the developer next demonstrates the game's behavior. Gamut essentially uses the Stimulus/Response style of demonstration as pioneered in Demo [17]. In object-oriented terms, a stimulus is represented by an *event* object, and the response is represented by *actions*. Events come from external sources such as pushing a button, moving the mouse, or a ticking of a timer. Actions are pieces of code which change the state of the application. Moving an object or setting the score are actions.

Gamut uses a simplified demonstration technique which we call *nudges*. A *nudge* is a correction that the developer gives the system when it is not behaving correctly. Gamut defines two nudges: the *Do Something!* nudge and the *Stop That!* nudge.

The developer uses Do Something! to add new behavior. First, the developer causes an event to occur like pushing a button. When an object does not respond, the developer selects the object and pushes the Do Something! button. This tells the system that the developer is ready to show a new example concerning the selected object. Stop That! is a means for demonstrating negative examples. The developer selects objects that did the wrong thing and presses the Stop That! button. Stop That! performs undo on actions that affect the selected objects. One complication is that actions which perform their work on multiple objects sometimes need to be split up to affect only selected items.

In the water droplet example, the developer wants to make the droplet follow the arrow lines each time the timer ticks. The droplet begins at the top of the pyramid (see Figure 2). Pressing the Step button on the timer provides the timer event. At this point, the system does nothing because no behavior has yet been demonstrated. The developer selects the droplet and pushes the Do Something! button. The system changes the state to Response mode and brings up a dialog asking the developer what to do next.

Response Mode

The developer uses Response mode to demonstrate new behavior. In this phase, Gamut acts like any other PBD system and records all modifications the developer makes to the state of the application. During Response mode, Gamut adds demonstration interactions to the interface (see Figure 3):

- Gamut shows the differences in all modified objects by making "temporal ghosts." Temporal ghosts are shown as dimmed versions of objects as they originally appeared.
- Hint highlighting is activated. The developer may highlight objects pertinent to the example that may not be apparent to the system by selecting them with the right mouse button. Highlighted objects become important hints that direct the inferencing. The hint highlighting interaction is separate from the selection widget interaction in Gamut to make highlighting easier and because selection is needed for other operations such as moving objects and changing their colors.

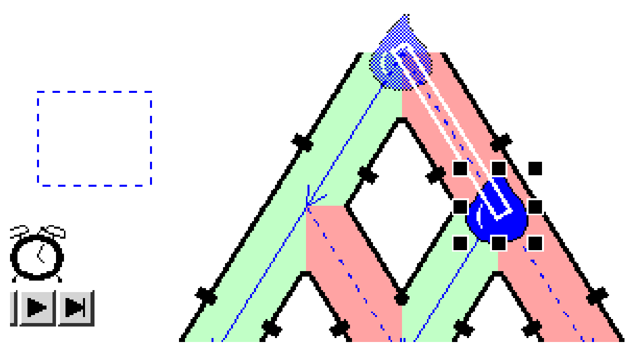

Figure 3: State showing the temporal ghost of the droplet at the top and a highlighted arrow.

Continuing with the example, the developer moves the droplet to the end point of the dotted arrow line. The system creates a ghost of the droplet at the beginning of the arrow (see Figure 3). The developer highlights the path that the droplet follows and presses Done to complete the response. Let us suppose that the developer did not highlight the dotted rectangle. When important state is not highlighted, the system must eventually query the developer.

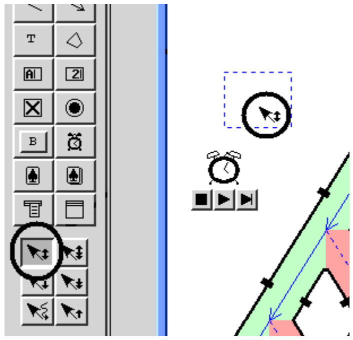

Figure 4: Using the mouse icons to show a click event. The circles indicate the palette choice and the click icon.

The developer next changes the style of the rectangle to solid and begins to demonstrate moving the droplet along the opposite path. When the timer Step button is pushed, the system (not knowing any better) moves the droplet along the dotted path. The developer selects the droplet and pushes the Stop That! button. Immediately, all actions performed on the droplet are undone moving it back one step to where it started. The developer corrects the action by dragging the water droplet to the solid side and presses Done. This confuses the system because it has no criteria to choose between the two paths and it asks the developer for a hint. Gamut stays in Response mode until the developer satisfies all of its questions. The developer is no longer allowed to make changes in the recorded response, but ghost objects and hint highlighting remain as a way to answer the dialog queries.

The developer has three ways to answer a question dialog:

- Highlight one or more objects and press the Learn button. This builds the highlighted objects into new relationships.
- Press the Replace button to eliminate the original response and replace it with the current response. This is used to correct mistakes.
- Press the Wrong button to skip the question. Since the system uses heuristics, it can sometimes create erroneous questions. The Wrong button forces the system to choose a different question which will hopefully be better.

The correct choice in this case is to highlight the rectangle and press Learn. Testing further, the developer finds the droplet now moves as desired.

Mouse Input

The developer now demonstrates how to toggle the rectangle. Assume that the rectangle begins in the dotted style. The desired interaction is to have the player click the mouse to switch the rectangle's style, so the developer drops a "mouse click" icon onto the window (shown in Figure 4). The mouse icons are used to show what the player will do with the mouse at run-time. Nothing happens originally so the developer pushes the Do Something! button, and changes the rectangle's style to solid. After pressing Done, the developer drops another click icon, and the system responds by setting the rectangle to solid (making it appear to do nothing). The developer selects the rectangle, presses Do Something! and changes the style back to dotted. This causes the system to question the developer asking what the style of the rectangle depends on. The developer highlights the ghost of the rectangle indicating that its previous style is the state that matters and presses Learn. Dropping further click icons causes the rectangle to act correctly.

Finishing up, the developer moves the water droplet back to the top of the pyramid. The guide objects are made invisible using a menu command. The droplet is started animating by pressing the play button on the timer widget. And finally, the mouse interaction is enabled by pressing a toggle button on the simulated window. The developer can click the mouse and see that the droplet follows the correct path.

GAMUT'S INFERENCING ALGORITHM

Most previous PBD systems have used very simple algorithms for inferring behavior, but these are not sufficient to handle the behaviors needed by real applications. There are a rich variety of algorithms developed by AI researchers and Gamut adapts two of them. Gamut uses *heuristic search* as the backbone of its inferencing. Heuristic search learns behavior quickly because rules encode useful features of the domain. Gamut also uses *decision tree learning* [14] which we describe later, but Gamut does not use the decision tree algorithm to produce code directly. Expressions which use a boolean test encode the test as a decision tree.

Gamut uses three kinds of objects to represent application behavior: *events*, *actions*, and *descriptions*. These components are similar to the language invented by Halbert in his SmallStar system [4]. An *event* object is the same as an Amulet "command object" [12]. Command objects are stored in widgets and input objects (called interactors). In some systems, a command object would be likened to a "callback procedure." A command object stores just enough state to implement Undo and Redo for the operation it performs which the system then copies to the undo history. Gamut augments command objects (a.k.a. events) by adding *actions* demonstrated by the developer, and then events become *behaviors*. As previously mentioned, actions are objects whose methods change the state of the application. Gamut defines six actions which are listed in Table 1.

Create Object: Creates one or more graphical objects from a prototype.

Delete Object: Removes graphical objects from their window.

Move/Grow: Moves or resizes graphical objects.

Change Property: Changes one property of one or more graphical objects.

Reparent: Moves one or more graphical objects from one window or group to another.

Change Z-Order: Changes the stacking order of one or more graphical objects (like To Top or To Bottom).

Table 1: List of all actions defined in Gamut.

The parameters of actions are called *descriptions* which are named after Halbert's "data descriptions." A description can be a constant value like a number or a color, or it can be a short, composable expression which is represented by an object. Description objects can have parameters which themselves are descriptions. For instance, “the color of the object to which the arrow points” is a Get Property description whose object parameter is a Connect description. Descriptions have two significant methods. One is a computation method which evaluates the description’s expression. The other is the “recursive difference” method which is used for revising the description. Gamut presently defines 11 descriptions which are listed in Table 2.

Get Property: Returns the value of a single property from a graphical object.

Align: Returns a graphical location based on connections to other objects and locations.

Select Object: Picks one or more objects from a set of objects according to a boolean expression.

Choice: Picks one value from a set of several values according to a boolean expression. This is Gamut’s form for an if-then or case statement.

Equal: Tests if two values are equal.

Number Test: Tests two numerical values for equality, lesser, or greater than.

Connect: Given a location, return the objects that are connected to that location.

Chain: A Connect that is repeated an integer number of times and returns the last set of objects.

Count Objects: Returns the number of objects in a set.

Add: Returns the sum of two numerical values.

Get Prototype: Returns the base type of a graphical object.

Table 2: List of all descriptions presently implemented in Gamut.

The Stages of Inferencing

The steps in Gamut’s inferencing are shown in Figure 5. In the first stage, actions from the original behavior and the developer’s manipulations are converted into new actions. The second stage compares the new actions with the actions in the original behavior and generates a data structure that captures the differences between the two. In the final stage, the differences found in stage two are resolved to create a new behavior.

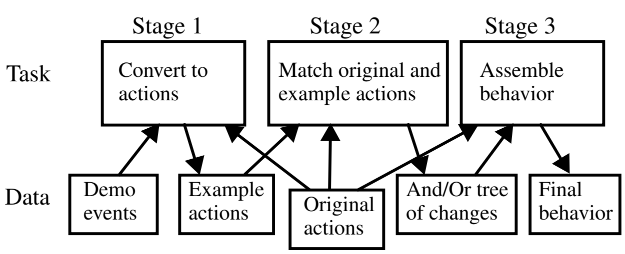

Figure 5: The stages Gamut uses to create new behavior.

Stage One

The first step converts the developer’s response into a list of actions. Gamut reads the undo history to extract the new example as a list of command objects (or events). The Do Something! and Stop That! commands along with the dialog’s Done button form the boundaries. If there was an original behavior, its actions are added to the list. Many graphical operations are complicated. For example, grouping objects creates a new group object, changes the selected objects’ positions, and reparents the selected objects into the group. Gamut therefore translates the raw Amulet events into more primitive actions to make them easier to manipulate.

Stage Two

In this stage, Gamut compares the newly formed example actions with the actions found in the original stimulus event. This stage performs the bulk of the inferencing work. There are two parts to this stage. The first part matches the actions in the new example to the original actions. This will determine which changes were made to the original actions’ parameters, which original actions did not execute, and which example actions are new. Of course, if there are no actions in the original stimulus, stage two is skipped and the system moves to stage three.

The second part of stage two propagates the changes found in the original actions’ parameters to the description objects contained within. Each description determines how its parameters change using its *difference* method. This is recursively repeated for the whole tree of descriptions. The final result of this stage is a set of changes one might make to the parameters of the original actions and descriptions in order to make it perform the same operation as the actions in the example. The algorithm Gamut is using is a heuristic search, and the object-oriented design provides a good framework to apply heuristics efficiently and allows extensibility.

Matching Actions

Here Gamut matches actions in the new example to actions in the original behavior. We intentionally designed Gamut’s actions to be unordered. Most systems use temporal order to represent constraints between actions. The result of one action feeds into the next to provide a combined result. Instead, Gamut uses explicit descriptions where what could be the result of another action is described fully by the action that actually needs it. Whether the actions execute before or after each other does not matter. Gamut prevents code duplication by sharing common descriptions. By being unordered, actions can be easily matched by looping through all the actions and comparing them. In our experience with Gamut, all behaviors contain very few actions (usually two or three), so the time to compare them is insignificant.

Two actions are considered identical if all of their parameters match. If all but one parameter matches, the actions are considered similar. Identical actions are ignored since they already do what is required. Similar actions are recorded as having changed the one parameter. Unmatched actions from the new example are considered to be new, and unmatched actions from the original behavior are considered not to have been executed.

There are a few complications to consider. The first concerns actions that operate on sets of objects. In this case, example actions which act on single objects must be coalesced with original actions which have that object as a member of its set. Only after all actions are matched can the system determine if all objects in the original action are accounted for and whether the original action is identical to the example.

A second complication concerns the Create Object action. If the developer demonstrates creating graphical objects, those objects will not be the same as objects created by the original Create Object action. Furthermore, objects created by a Create Object action might be deleted by the developer and replaced by identical objects. Instead of matching the parameters of Create Object actions, the system examines the graphical objects created by the developer to see if they "look the same" as the prototype objects from which a Create Object action instantiates new objects. This involves examining the graphical object's properties and type to see if they match. The prototype parameters of Create Object actions match when they produce objects which look alike.

Propagating Changes To Descriptions

Once a potential change to a parameter is determined, it is propagated to the description stored in that parameter. If the old description is constant, Gamut records that the constant may have changed to the new value. Otherwise, the system calls the description's *difference* method.

The difference method takes the new value and tries to find a way to modify the description to generate that value. Each type of description has its own heuristic difference method, but most simply apply what we call the "one change" rule. Using the one change rule, an algorithm only tries to change one thing. This is similar to Winston's concept of a "near miss" for training examples [16]. The difference method tries changing one of the description's parameters at a time leaving the others the same. For instance, the Get Property description retrieves a property from a graphical object. It has two parameters, the object and the name of the property to get. Using the "one change" rule, the difference method will first search for a different object whose property has the right value. Next, it will look for different properties in the original object to find the value. All matches that it finds are added to the set of changes. The new values are then recursively propagated to the Get Property description's parameters. A few descriptions such as Align (which computes the location of graphical objects) use more complicated heuristics, but they achieve similar results. Propagating new values through the original behavior is common in AI techniques like reinforcement learning and neural networks but has not been previously applied to a PBD tool.

The developer is normally not queried during the search process. Also, most difference methods do not use hinted objects because they are sufficiently constrained by the "one change" rule not to produce too many results. An exception to this is the Count Objects description which returns the number of graphical objects in a set. This description cannot be computed in reverse since knowing that the description is supposed to compute 5 instead of 3 does not tell the description which 5 objects to count. Instead, it asks the developer to highlight the objects and proceeds using that answer.

If the difference methods are poorly designed, there is a potential for a combinatorial explosion. The depth of the search is fixed to be the depth of the description hierarchy, but the number of changes that each description can find could be high. The one change rule and the structure of the application will normally restrict the search space adequately. However, some difference methods must cull the amount of changes they find to only those most likely to be the ones the developer means. Because each description has its own difference method, it can be tailored to meet the specific needs of that expression.

Because the parameters of Gamut's descriptions are not independent, the result of the difference methods form an And-Or tree. The leaves of the tree are the parameter changes found by the methods. And-nodes indicate when all changes must be made to correct the description and Or-nodes choose among a set of optional changes. Note that this And-Or tree is not related to the decision tree algorithm that is discussed later.

Stage Three

After completing the analysis stage, the system is ready to assemble the new behavior. Actions that were judged identical or similar to the example are kept. New actions are added in and since order does not matter they can be appended to the end of the list. Unmatched actions are enclosed in a Choice description (see Table 2) to indicate that there is a condition. If the behavior already contained Choice descriptions, Gamut uses the matching results to pick the best branch and proceeds to modify that.

What remains is to resolve the set of changes found by the difference methods, and to replace the constant parameters of new actions with descriptions if warranted. First, Gamut will search the old behavior for descriptions that already produce the desired result. Descriptions are often shared among actions and searching prevents the system from having to learn descriptions twice. If the search fails, Gamut will try to create a new description. Gamut uses the same algorithm as Marquise [13] to accomplish this task. However, instead of using only heuristics to find matching descriptions, Gamut focuses attention on the highlighted objects.

Creating a new description involves picking one whose type can evaluate to the correct kind of value. Beyond that, the system uses heuristic rules to pick highlighted objects that can be used as parameters. When Gamut replaces the original description, it creates a Choice description and stores both the new and old description together. By not referring to the old value's branch in the Choice's boolean expression, it remains uncomputed, but is available for comparison with new descriptions. Gamut uses these "dead" branches to prevent itself from repeating incorrect inferences.

If Gamut cannot create a new description using the highlighted objects, it asks the developer to highlight different objects. Gamut uses the context of the change including the affected action, the description that is actually modified, and the old and new values, to form an English sentence. From this sentence, the developer can choose to highlight new objects (which are processed as before), or choose to replace the changed value with the new value (without creating a new description), or move on to another question. Any choice the developer makes serves to eliminate items from the set of changes. The system will continue posing questions until the set is empty.

DECISION TREE LEARNING

So far, we have only mentioned the decision tree algorithm in passing. In Gamut, decision trees are used to represent boolean expressions in two kinds of descriptions: the Choice description and the Select Object description. Each of these descriptions keeps data to support its own local decision tree.

The decision tree algorithm was invented by Quinlan [14] as a form of inductive learning (Gamut uses the generic ID3 version of the algorithm). The data from which a decision tree learns has a database-like structure. Each row of the database provides an example of the concept to be learned, and each column represents an attribute of the concept. For instance, the concept might be whether the weather is good for a picnic. Attributes of the weather might be the temperature, whether it is raining, and the humidity. The diagram in Figure 6 shows a possible result of converting a database containing these data into a decision tree.

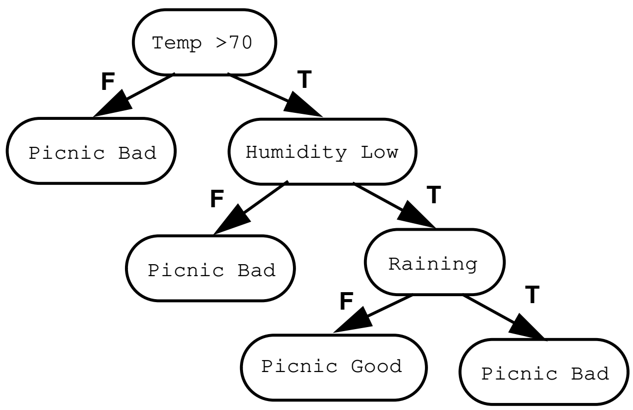

Figure 6: Decision tree to gauge picnic weather conditions.

The decision tree algorithm uses a statistical measure called "entropy" that relates how well an attribute corresponds with a result. The algorithm chooses the attribute with the best entropy value to be the root of the tree, eliminates that attribute from the database, and recurses for each branch of the tree. In Gamut, decision trees learn which objects the Select Object description and which branch the Choice description should pick. Gamut's decision trees never grow very big because most boolean expressions it needs combine only a few attributes. An example attribute might be "an object has a dotted line style."

When a Select Object or Choice description is first created, Gamut creates attributes for its database. The attributes are just descriptions and are constructed from highlighted objects as described above. Gamut scans the highlighted objects to create predicates from their properties and positions. Gamut adds new predicates whenever it finds that two examples in the database contradict each other.

Each time a Select Object or Choice description is revised, the system will first try to add a new example row to the decision tree database. New examples are formed by evaluating the set of predicates and pairing that with the new result indicated by the revision. If the new example is different from the other examples, a new decision tree can be generated which will distinguish the new case. If the new example contradicts another, though, more predicates will need to be created.

Gamut's use of decision trees is considerably different from the standard practice. Normally, decision trees use handcrafted attributes and hundreds of examples to learn a complicated expression. Gamut uses automatically generated attributes and a few examples to learn simple expressions. The entropy measure serves as a good way to pick out the one or two attributes that matter out of a set of dozens.

RELATED WORK

A number of other systems build programs by demonstration. Previous systems have always relied on the developer to manually edit the inferred code in order to add conditions and correct mistakes that the system could not handle.

The system probably the most like Gamut is our earlier system, Marquise [13] which was designed to build graphical editors such as node and line diagrams. Gamut's behavior structure as well as the means for generating descriptions were taken from Marquise. Also, Gamut's mouse arrow icons for demonstrating mouse events came from Marquise. The guide object ideas in Gamut were first used in Demo II [1], Metamouse [8], and Rehearsal World [3] as was previously mentioned.

Grizzly Bear [2] has a good inferencing system for finding linear constraints and some forms of conditions. It also had an algorithm for recognizing certain groups of objects. For instance, it could learn the set of all objects whose color is blue. Moving beyond that level of description, though, was not possible. Furthermore, behaviors could not be refined through demonstration. Grizzly Bear required all examples for a given behavior to be demonstrated at once forcing the developer to devise examples very carefully. Grizzly Bear's inferencing was primarily for learning linear constraints between the positions of objects which, for instance, could not learn to make an object follow a chain of lines. The Pavlov [18] system also inferred linear constraints, and it also had a partial guide object mechanism for making behaviors where objects move and turn. However, conditional relationships such as modes and input events could only be added by annotating the code.

The Cima [9] system was one of the first PBD systems to incorporate inductive learning algorithms designed by AI researchers. Cima was designed to learn word and letter combinations that are commonly found in word processing tasks. It used hints (but no guide objects), and it also used positive and negative examples. Unfortunately, Cima was never truly finished. It was never applied to actually perform word processing tasks and could not infer actions or behavior. Furthermore, the manner in which Cima's inferencing was applied to its textual domain makes it difficult to re-engineer for another domain.

FUTURE WORK

Gamut is implemented using Amulet [11] and runs on Unix, Windows, and the Macintosh. At the time of this writing, Gamut is still not quite complete. We have made significant progress and can now create many complex applications, but it still lacks the polish developers need to make a system usable. The current work involves improving Gamut's feedback and filling out the features to make more applications possible. We want potential developers and our eventual test subjects not to find implementation holes too easily.

A major unanswered issue is how to properly display behaviors that the developer demonstrates. Even though the developer can completely test and revise behaviors through demonstration, it is still important that the developer be able to examine what the system has learned. People may be able to see gaps and mistakes in the behavior even if they do not know how to write code in the system's language. The feedback used by other systems such as the comic book metaphor used in Pursuit [10], are not applicable to Gamut because they are meant to display behaviors with many actions but simple parameterization. Gamut's behaviors have few actions but complex parameterization.

Finally, we need to complete a formal user test. We have been performing informal tests to find the larger usability bugs and to help steer the project, but as a final step, we intend to show that nonprogrammers can actually build complete games using Gamut.

CONCLUSION

We are quite excited about the potential shown by Gamut's inferencing algorithm. Gamut can already make whole applications solely through programming by demonstration. So far, the applications are at the level of Tic-Tac-Toe and a Turing machine, and we have created behaviors such as a monster that moves like a monster in PacMan. Soon, Gamut will have enough features to build even more games.

The algorithm works through a combination of several interwoven features. First, the developer can create guide objects which make visible the factors that influence behaviors. Second, the developer can point out objects as hints during demonstration to tell the system on which objects a behavior depends. This significantly reduces the amount of search the system must perform to generate and revise the inferred code. Third, the system uses a straightforward "one change" heuristic to find the differences between a new example and the behavior it is meant to revise. The difference methods can find specific changes deep inside the behavior which will make it produce the right results. Finally, the algorithm incorporates decision tree learning to handle concepts that its heuristic search portions cannot understand. This algorithm makes it possible to infer conditions with objects that are not directly affected by the actions which depend on them.

We think that the kinds of learning performed in Gamut could find uses outside of the board game domain. The Stop That! and Do Something! style of interaction seem to be quite intuitive and could be used to perform other tasks such as programming macros. The inferencing algorithm could be extended to different domains by changing the description language to include different concepts. We believe that Gamut even in its current, unpolished, state shows a significant improvement in PBD methods. This system will help bring computing power ever closer to those nonprogrammers who want to apply their skills to make software.

ACKNOWLEDGEMENTS

This research was partially sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326, and partially by NSF under grant number IRI-9319969. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

REFERENCES

1. G.L. Fisher, D.E. Busse, D. A. Wolber. *Adding Rule-Based Reasoning to a Demonstrational Interface Builder*. Proceedings of UIST'92, pp. 89-97.
2. M. Frank. *Model-Based User Interface Design by Demonstration and by Interview*. Ph.D. thesis. Graphics, Visualization, & Usability Center, Georgia Inst. of Tech., Atlanta, GA, 1996.
3. L. Gould, W. Finzer. *Programming by Rehearsal*. Palo Alto Research Center, Xerox Corporation, 1984.
4. D.C. Halbert. *Programming by Example*. Ph.D. thesis, Computer Science Division, EECS Department, University of California, Berkeley, CA, 1984.
5. *Hypercard*. Apple Computer Inc., Cupertino, CA, 1993.
6. F. Lionet, Y. Lamoureux. *Klik & Play*. Europress software, 1996.
7. Macromedia. *Director.* 600 Townsend Street, San Francisco, CA 94103, macropr@macromedia.com, http://www.macromedia.com/ 1996.
8. D. Maulsby. *Inducing Procedures Interactively: Adventures with Metamouse*. Masters thesis. Research Rept. 88/335/47. University of Calgary, December, 1988.
9. D. Maulsby. *Instructible Agents*. Ph.D. thesis. Dept. of Computer Science, University of Calgary, Calgary, Alberta, June 1994.
10. F. Mudugno, T.R.G. Green, B.A. Myers. *Visual Programming in a Visual Domain: A Case Study of Cognitive Dimension*. Proceedings Human-Computer Interaction'94, People and Computers, Glasgow, August, 1994.
11. B.A. Myers, R.G. McDaniel, R.C. Miller, A. Ferrency, A. Faulring, B.D. Kyle, A. Mickish, A. Klimovitski, P. Doane. "The Amulet Environment: New Models for Effective User Interface Software Development," *IEEE Transactions on Software Engineering*, to appear. http://www.cs.cmu.edu/~amulet
12. B.A. Myers, D.S. Kosbie. *Reusable Hierarchical Command Objects*. Human Factors in Computing Systems, Proceedings SIGCHI'96, Denver, CO, April, 1996, pp. 260-267.
13. B.A. Myers, R.G. McDaniel, D.S. Kosbie. *Marquise: Creating Complete User Interfaces by Demonstration*. Proceedings of INTERCHI'93: Human Factors in Computing Systems, 1993, pp. 293-300.
14. J.R. Quinlan. *Induction of Decision Trees*. Machine Learning, Kluwer Academic Publishers, Boston, Vol. 1, 1986, pp. 81-106.
15. D.C. Smith, A. Cypher, J. Spohrer. *KidSim: Programming Agents Without a Programming Language*. CACM, Vol. 37, No. 7, July 1994, pp 54-67.
16. P.H. Winston. *Learning Class Descriptions From Samples*. Artificial Intelligence, Chapter 11, Addison-Wesley Publishing Company, Reading, MA, 1984, pp 385-408
17. D. Wolber. *Developing User Interfaces By Stimulus Response Demonstration*. Ph.D. Thesis, Computer Science Department, University of California, Davis, 1992.
18. D. Wolber. *Pavlov: Programming By Stimulus-Response Demonstration*. Human Factors in Computing Systems, Proceedings SIGCHI'96, Denver, CO, April 1996, pp. 252-259.