A Pure Reasoning Engine for Programming By Demonstration

Martin R. Frank *James D. Foley*

Graphics, Visualization & Usability Center
Georgia Institute of Technology
Atlanta, Georgia 30332-0280

(404) 853-9382 (404) 853-0672
{martin,foley}@cc.gatech.edu

ABSTRACT

We present an inference engine that can be used for creating Programming By Demonstration systems. The class of systems addressed are those which infer a state change description from examples of state [9,11].

The engine can easily be incorporated into an existing design environment that provides an interactive object editor.

The main design goals of the inference engine are responsiveness and generality. All demonstrational systems must respond quickly because of their interactive use. They should also be general - they should be able to make inferences for any attribute that the user may want to define by demonstration, and they should be able to treat any other attributes as parameters of this definition.

The first goal, responsiveness, is best accommodated by limiting the number of attributes that the inference engine takes into consideration. This, however, is in obvious conflict with the second goal, generality.

This conflict is intrinsic to the class of demonstrational system described above. The challenge is to find an algorithm which responds quickly but does not heuristically limit the number of attributes it looks at. We present such an algorithm in this paper.

A companion paper describes Inference Bear [4], an actual demonstrational system that we have built using this inference engine and an existing user interface builder [5].

KEYWORDS: Programming by Demonstration.

INTRODUCTION

When one creates an interactive graphical application today, one typically uses an interface builder as the first step. Interface builders are easy to use - they are more like structured drawing editors than programming environments. The much harder part of building a graphical application is specifying the *behavior* of the user interface - it typically involves writing a textual specification in a scripting or programming language. The formal nature of this process excludes a non-technical audience from specifying behavior. This is unfortunate, especially because most of the desired functionality at the user interface level is simple, such as making user interface elements align, resize, center, appear, change color, and so on.

This problem is well suited for a Programming By Demonstration approach which uses examples of state for inferencing. This is because the users already know how to change the state of the interface using the interface builder. They can then demonstrate how user interface elements are created, deleted and modified in response to run-time events.

Programming By Demonstration is a young discipline which has just recently emerged as a recognized subfield of user interface software [2]. This becomes apparent when one compares the prototypes that have been built - the most striking observation is that each one uses a unique approach and that most are tightly coupled to the domain they were built for.

Separate demonstrational systems for separate tasks are better than none at all, but is there some common ground? Can one build a demonstrational inference engine which can be used for a class of demonstrational systems?

The benefit for the builders of Programming By Demonstration systems would be that they do not have to start from scratch - they can start with the generic inference engine and then tune it later.

Even more importantly, the benefit for users of these demonstrational systems would be that they can rely on a common methodology for demonstrating behavior.

We present such an engine which addresses a subclass of demonstrational systems. This subclass consists of systems that infer a generalized state change description given several examples of state. Existing demonstrational systems that fall into that category are DEMO [11] and the Geometric Interactive Technique Solver [9]. The constraint solver of Chimera [6] roughly falls into this class, but has no notion of time - it uses “valid” states rather than “before” and “after” states.

This is “Frank, M. and Foley, J. A Pure Reasoning Engine for Programming By Demonstration. In *Proceedings of UIST’94,* ACM Symposium on User Interface Software and Technology, (November 2-4, Marina del Rey, California) ACM, New York, 1994”.

Demonstrational systems not addressed by our engine include those that use a domain-specific rule base to guess relationships between objects, such as Peridot [8] and Druid [10]. These systems can often infer relationships from a single example.

Other systems that are not addressed are those that automate repetition by watching the user, such as Eager [1] and Metamouse [7]. AIDE (in [2]) is a proposed domain-independent framework for this class of demonstrational system.

PHILOSOPHY OF THE INFERENCE ENGINE

Our inference engine is based on the following principles.

It contains no domain knowledge. The design goal of the inference engine is to be useful for a range of application domains. Consequently, we cannot base its inferencing on knowledge about a particular domain. The disadvantage is that we cannot make use of such knowledge to aid the inference process. The advantage is that our engine can be used for many domains. It can also be used at any level of abstraction. For example, it can be used to demonstrate how a dragged object follows the mouse pointer, but it can also be used to demonstrate that the number of employees increases by one if a new employee is hired.

It does not guess. A specialized demonstrational system can incorporate domain knowledge to help it make inferences. For example, Peridot [8] has built-in knowledge about the concepts of centering and aligning. This allows it to guess the user's intention when they are centering elements, making the system easy to use. Our engine is an experiment on how far we can push a domain-independent demonstrational system without compromising ease of use.

It finds the simplest possible solution. This is again in contrast to rule-based systems which can infer that a new object should be centered between two existing objects even when it is given only a single example. In this case, our engine would initially infer that the new object is created at that exact location.
The advantage of finding the simplest possible solution is that the inference engine is predictable - it behaves consistently for each demonstration.
The advantage of finding the most likely solution using a domain-dependent rule base is that it requires fewer examples (often only one). The disadvantage is that the engine may appear erratic to the user - it may produce different solutions for apparently similar situations because different rules fire, and it may sometimes fail to make a computationally simple inference because no rules fire.
The best solution may be to combine both approaches by first checking if domain-specific rules fire and by then falling back on a domain-independent reasoning engine such as ours. At least one demonstrational system has used such a hybrid approach [3].

The solution is correct for the given examples. Correct here means that the resulting generalized state change description will indeed work for each of the examples you provided. "It is correct" in this sense is not to be confused with "it is what you intended" - it is possible that the system will respond with an unexpected solution. In this case, you have to provide one or more additional examples which contradict the current solution but are consistent with the desired solution.

It can infer changes to any attribute. The engine can infer assignment of a constant (a.color := "blue") and assignment from another variable (a.color := b.color) even if it does not know about the type of the attribute. It can make more advanced type-specific inferences if it does (a.x := a.x + 1/2 * a.width).
The above statement should not be confused with "it can infer changes to any attribute *of an arbitrarily complex nature*" - all (successful) demonstrational engines look for simple relationships[1].

It is extensible. The engine comes with three predefined inference types: string, integer, and boolean variables. Inference types test how a variable can be computed from other variables. For example, a string variable may be computed by concatenating two other string variables, a boolean variable may be computed as the logical negation of another boolean variable, and so on.
Other inference types can easily be added. For example, you can define a "color" type by providing code that describes how a color variable can be computed from other color variables.

TERMINOLOGY

The input to the inference engine consists of one *demonstration*. A demonstration can consist of a single example for simple behavior or two or more examples for more complex behavior. An *example* consists of a Before snapshot, a parameterized event, and an After snapshot. The event is the "stimulus", the After snapshot the "response" in the terminology of [11] (the Before snapshot provides context).

The inference engine generalizes from the examples and returns a *script*. A script describes a change of state triggered by an event. Scripts can create, delete and change objects. A script is said to *solve* a demonstration if it transforms each of its Before snapshots to the corresponding After snapshots.

COORDINATE SYSTEM

We use a simple pixel-based coordinate system to describe the location and size of user interface elements (rather than a resolution-independent coordinate system such as constraints). The rationale is to not limit the use of our inference engine to the (few) interface builders that support resolution independence. Using a pixel-based coordinate system is the lowest common denominator - all interface builders we are familiar with can export object information using pixel coordinates.

OBJECTS, EVENTS AND SCRIPTS

The inference engine is based on the abstract data types Object, Event and Script. The environment making use of

1. An Artificial Intelligence subfield called "Automatic Programming" failed precisely because it tried to infer arbitrary programs from examples of input and output.

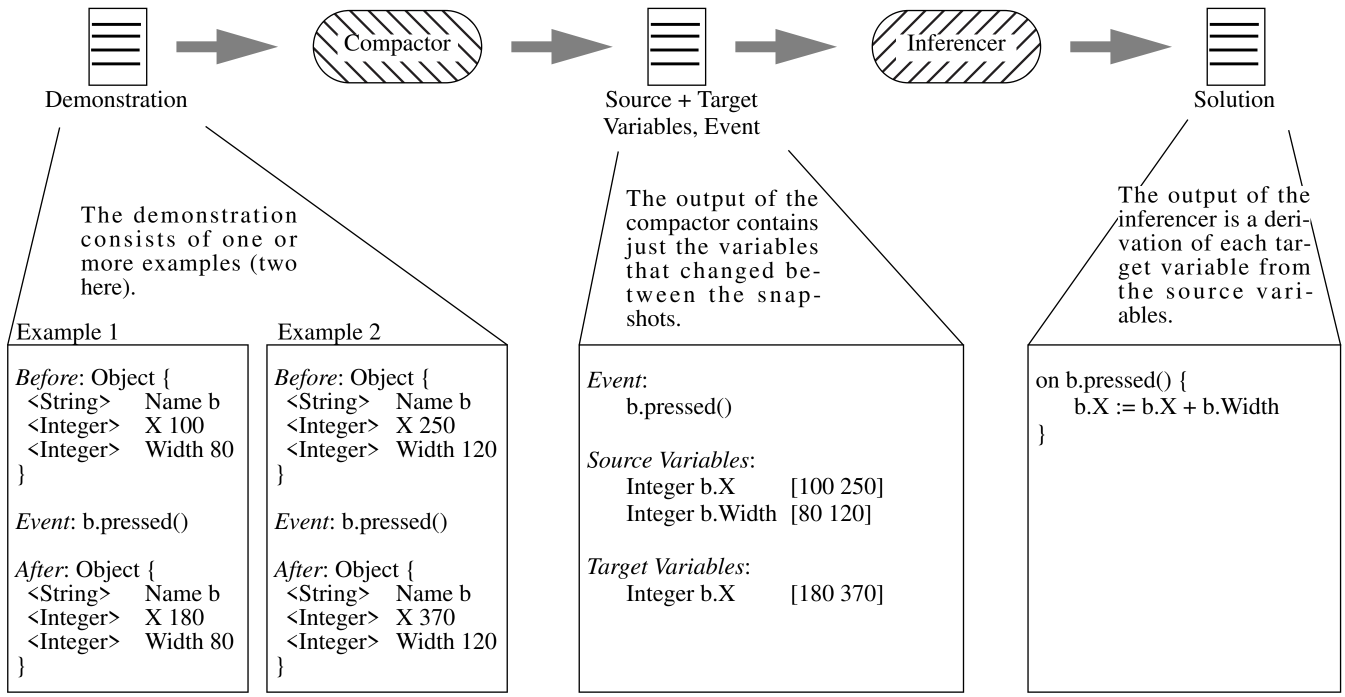

Figure 1. An Introductory Example.

our engine may or may not be based on these elements. If it is not, it maps its objects and events into this format, hands them to the inference engine, and then translates the resulting script into its native language.

Objects

Objects are a collection of attributes. Attributes of an object have a name, a type, and zero or more values.

```
Object {
    Height:      <Integer>     140
    Selected:    <Boolean>     1
    Type:        <String>      Container
    Width:       <Integer>     260
    X:           <Integer>     33
    Y:           <Integer>     32
    children:    <String>      OnButton OffButton
    mapped:      <Boolean>     1
    name:        <String>      container
}
```

The “name” attribute of an object has no special status. Its value can be derived from other attributes just like any other attribute of type String.

Events

From the engine’s perspective, an event is identified by an event name, such as “moved”, and by the name of the interface object on which the event occurred, such as “button”. This assumes that there is a layout hierarchy of interface objects so that all events that occur in a window are associated with an object (possibly the root object). Events can also have parameters.

```
button.moved(Integer x 40, Integer y 60)
```

The inference engine does not contain knowledge about the type of events that could occur. It simply treats the given event as the trigger for the behavior it infers. In addition, its event parameters can become parameters of the inferred behavior.

Scripts

Scripts describe how the state changes given an event. A simple example is given below.

```
on button.moved(Integer x, Integer y)
{
    line.x1 := x + 1/2 * button.Width
    line.y1 := y + button.Height
}
```

INTRODUCTORY EXAMPLE

Figure 1 shows a small but complete example of the inferencing process. The user has given two examples of a button moving one button length to its right in response to clicking on it.

The inferencing is done in two stages. In the first phase, the “Compactor” reduces the amount of objects and attributes that the inferencing process has to be concerned with. This is done by eliminating all objects and attributes which remain constant in the examples. The result of the Compactor is a list of “source” variables and a list of “target” variables.

Source variables are potential parameters of a solution. The compactor identifies attributes as source variables if they change between any two Before snapshots.

Target variables are the variables that have to be solved. The compactor identifies attributes as target variables if they change from any Before snapshot to a corresponding After snapshot.

The source and target variables are the input of the "Inferencer" which tries to derive each target variable from the source variables and from constant values. If it succeeds, it produces a script which contains an assignment for each target variable.

Inferencing - the search for relationships between variables - is inherently expensive. The Compactor eliminates irrelevant information so that the computationally much more expensive Inferencer is given the least possible information. In that way, inferencing is efficient even for user interfaces which contain many objects. In the above example, there could be many other interface elements besides the button for which functionality is demonstrated but the input to the inferencer would remain the same if the other elements are not touched during the demonstration.

THE COMPACTOR

There are two ways to find out which variables changed between snapshots.

Ideally, the interface builder allows us to be notified of each individual change that the user makes when preparing the snapshots. In this case, we can directly find out about changes between snapshots, which would make the Compactor obsolete.

Unfortunately, many interface builders do not allow external access to individual modification events. However, all of them are able to export information about the complete current state - the lowest common denominator is a file format for storing and retrieving designs. The Compactor efficiently recovers what the user has changed between snapshots from these complete states.

Source variables are those that changed between any two Before snapshots. Source variables take their values from the Before snapshots. The Compactor identifies them by the following process.

It first constructs a vector of values for each attribute. This attribute becomes a source variable if the vector's minimum value is not approximately equal to the vector's maximum value. "Approximately equal" is defined for each type of attribute. By default, strings are required to be exactly equal but screen coordinates are allowed to differ by up to fifteen pixels.

In Figure 1, the value vector of attribute *X* is [100 250] in the Before snapshots. It becomes a source variable because 100 is not approximately equal to 250. The attribute *Name* does not become a source variable because the elements of its value vector, [b b], are approximately equal.

Identifying source variables is linear in the number of attributes in the Before snapshots assuming that accessing attributes by name takes constant time (hash-based access).

No attribute can become a source variable if the demonstration consists of a single example because a single value is always approximately equal to itself. This is intended - there is no point in designating source variables because the Inferencer, described below, will always solve single-example demonstrations by assigning constant values to the target variables.

Target variables are those that change from any Before snapshot to a corresponding After snapshot. These variables take their values from the After snapshots. The Compactor constructs target variables in two phases - it first identifies target variables and then collects their values.

The Compactor identifies attributes as target variables by comparing the value of each attribute in a Before snapshot to its value in the corresponding After snapshot. The attribute is added to the target variable list if the two values are not (exactly) equal. The Compactor constructs the value vectors once all target variables have been identified.

In Figure 1, the attribute *X* becomes a target variable because its value changes from a Before to a corresponding After snapshot. For example, it changes from 100 to 180 in the first example. Attribute *Width* does not become a target variable because it never changes its value in response to an event. It remains 80 in the first example and 120 in the second example.

THE INFERENCER

The Inferencer is the component which relates target variables to source variables.

The Inferencer first groups the source and target variables by type. It then tests each target variable against unordered sets of source variables of the same type. A single such test checks if the target variable can be computed from a combination of the source variables. These tests are specific for each inference type. The Inferencer itself does not contain knowledge about types, it simply calls the test that is supplied by the inference type.

The size of the sets increases over time. Testing ends if (1) a test succeeds, (2) the target variable has been tested against all unordered sets, or (3) a type-specific limit on set sizes is reached.

In the demonstration of Figure 1, the Inferencer first tests the target variable b.X against the empty source variable set. A test against an empty set succeeds if the target variable can be solved by a constant (e.g. b.X := 100), which is not the case here. The Inferencer then tests against the single-member sets {b.X} and {b.Width}. These tests also fail. The test against the set {b.X, b.Width} succeeds as shown in Figure 1.

The Inferencer's running time is worst-case exponential in the number of source variables of the same type. This does not present a problem because no variable becomes a source variable unless it is changed by the user during a demonstration. In our experience, demonstrations which would bring the Inferencer to its knees (more than, say, fifteen source variables of the same type) do not occur in practice because they would require too much effort to demonstrate. Differently stated, scripts of such complexity are not suitable for a domain-independent Programming By Demonstration approach unless it is combined with other techniques.

PLUG-IN INFERENCE TYPES

The code which tests a target variable against a set of source variables is provided by the inference type. The engine comes with three common types: integers, booleans, and strings. The inference engine allows plugging in of additional inference types.

Defining an additional inference type is easy - all you have to do is subclass from the generic inference type by providing a new type name and a routine that tests when a target variable of that type can be computed from a set of source variables. We present one type in more detail.

THE INTEGER INFERENCE TYPE

A standard integer inference type comes with the engine. You can either use it for all variables of numeric type or you can create more specialized types such as "temperature" and "screen coordinate" by further subclassing from the integer type. By distinguishing between those integer types, the engine will not draw inferences that combine their values.

The standard integer type can derive a target variable from a linear combination of source variables. That is, given target variable t and source variables s_1 to s_n it can determine the relationship $t = c_1s_1 + c_2s_2 + ... + c_ns_n + c_0$ given n+1 substantially different examples.[1]

The algorithm that tests integer target variables against source variables works as follows. If the set of source variables is empty, the algorithm computes the arithmetic mean of the values of the target variable and tests if this constant is a solution.

For example, if the target variable has the values [18 17 22] in a three-example demonstration, the algorithm computes the arithmetic mean, 19, and then tests if the vector [18 17 22] is approximately equal to [19 19 19]. If it is, the algorithm has solved the target variable (t:=19).

If the set of source variables is not empty, the algorithm constructs a matrix and a vector that can then be solved by Gaussian elimination. Assume there are *n* source variables $s_1...s_n$. If there are less than *n* examples, the test fails. If there are exactly *n* examples, the algorithm tries to derive the target variable from the source variables without an additive constant ($t = c_1s_1 + c_2s_2 + ... + c_ns_n$). If there are more than *n* examples it tries to solve the general case ($t = c_1s_1 + c_2s_2 + ... + c_ns_n + c_0$).

This is again best explained through examples.

Example 1

Consider the introductory example of Figure 1, where the variables are as follows.

Source Variables:
Integer b.X [100 250]
Integer b.Width [80 120]

1. "Substantially different" is synonymous with "linearly independent" for the integer inference type.

Target Variables:
Integer b.X [180 370]

The algorithm takes these examples and constructs the following matrix. The columns of the matrix correspond to the values of the source variables (b.X and b.Width here), respectively. The vector is made up from the values of the target variable that we are trying to solve (b.X here).

$$\begin{bmatrix} 100 & 80 \\ 250 & 120 \end{bmatrix} \times \begin{bmatrix} \text{b.x} \\ \text{b.width} \end{bmatrix} = \begin{bmatrix} 180 \\ 370 \end{bmatrix}$$

Standard Gaussian elimination can then be used to solve this set of equations.

$$\begin{bmatrix} \text{b.x} \\ \text{b.width} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

That is, we have found a relation between the target variable and the source variables, namely

$$\text{b.X} \leftarrow 1 \cdot \text{b.X} + 1 \cdot \text{b.Width}$$

The inference engine then checks if this is indeed a solution by re-substitution[2]. A simplifier brings the solution to its final form, b.X ← b.X + b.Width.

All examples in this paper use demonstrations which can be solved exactly for the sake of simplicity. We discuss later how we use snapping to deal with approximate solutions such as "b.x =1.03, b.width = 0.98"

In the previous example, there were no redundant examples ("bad examples"), and there were only *n* examples for *n* variables. The following example shows how the algorithm constructs the matrix in a more general case.

Example 2

In this example, assume that the user has given several demonstrations that center interface object *b* between objects *a* and *c* by moving *b* (rather than resizing *b*). Figure 2 shows the variables relevant to the demonstration.

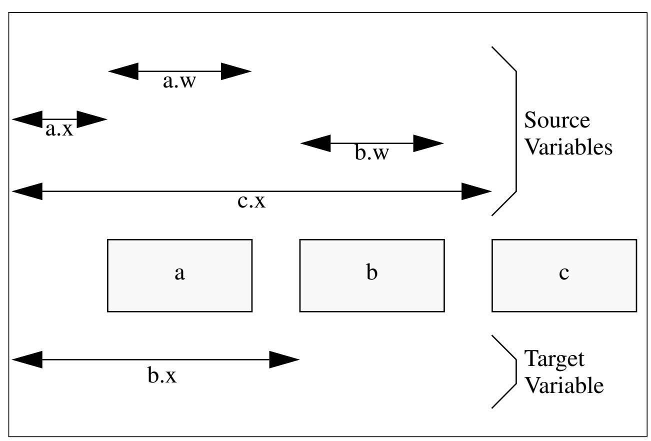

Figure 2. Centering Elements.

2. Re-substitution would not be necessary if the exact solution is used, of course. However, we also use re-substitution to test if a "snapped" version of the solution will do as explained later.

The formula for *b.x* which centers *b* in this way is shown below.

$$b.x \leftarrow a.x + a.w + \frac{1}{2}(c.x - (a.x + a.w)) - \frac{1}{2}b.w$$

Let us assume that the demonstration consists of seven examples as shown in Table 1.

Example:		1.	2.	3.	4.	5.	6.	7.
source variable	a.w	200	200	100	100	50	100	200
source variable	a.x	200	200	200	0	0	100	200
source variable	b.w	100	100	200	200	50	50	100
source variable	c.x	800	600	800	800	150	300	800
target variable	b.x	550	450	450	350	75	225	550

Table 1. Example values.

If we have more examples than needed, which ones should we select? Ideally, we want to use the most "unique" examples. Seen from a demonstrational standpoint, one intuitively feels that using nearly identical examples will not give the inference engine more useful information. Seen from a "math" standpoint, one does not want to have rows in the matrix that are nearly identical because that increases the likelihood that one row is linearly dependent on the others (that the augmented matrix is unsolvable).

We use the following method to select examples that will be used in the matrix[1]. We define the *distance* between two examples as the count of source variables that are not approximately equal between them. We define the *uniqueness* of an example as the sum of its distances to the other examples. We then select *k* examples of decreasing uniqueness for inclusion in the matrix (where *k* is the number of source variables plus one). Table 2 lists the distances for each example in Table 1.

Example:	1.	2.	3.	4.	5.	6.	7.
1.	-	1	2	3	4	4	0
2.	1	-	3	4	4	4	1
3.	2	3	-	1	4	3	2
4.	3	4	1	-	3	3	3
5.	4	4	4	3	-	4	4
6.	4	4	3	3	4	-	4
7.	0	1	2	3	4	4	-
Uniqueness:	14	17	15	17	23	22	14

Table 2. Computing the "uniqueness" of examples.

For example, the distance between the first and second example in Table 1 is one because *c.x* is the only source variable with a differing value. The distance between the first and fifth example is four because all four source variables differ between them.

Thus, we select examples two through six to construct the matrix below. In this demonstration, we have more examples than source variables which allows us to add the row of ones which tests for a constant offset (e.g. p.x := q.x + 100).

$$\begin{bmatrix} 200 & 200 & 100 & 600 & 1 \\ 100 & 200 & 200 & 800 & 1 \\ 100 & 0 & 200 & 800 & 1 \\ 50 & 0 & 50 & 150 & 1 \\ 100 & 100 & 50 & 300 & 1 \end{bmatrix} \times \begin{bmatrix} a.w \\ a.x \\ b.w \\ c.x \\ const \end{bmatrix} = \begin{bmatrix} 450 \\ 450 \\ 350 \\ 75 \\ 225 \end{bmatrix}$$

Solving the matrix returns

$$\begin{bmatrix} a.w \\ a.x \\ b.w \\ c.x \\ const \end{bmatrix} = \begin{bmatrix} 0.50 \\ 0.50 \\ -0.50 \\ 0.50 \\ 0 \end{bmatrix}$$

which results in the following derivation after running it through the simplifier. The formula indeed centers *b* such that it has equidistant edges to *a* and *c*.

$$b.x \leftarrow \frac{1}{2}a.w + \frac{1}{2}a.x - \frac{1}{2}b.w + \frac{1}{2}c.x$$

DYNAMIC OBJECT CREATION AND DELETION

So far, we have presented how the inference engine deals with single attributes by computing their value from other attributes. In this section, we explain how it deals with the run-time creation and deletion of objects.

After each demonstration, the engine first looks for objects that have been deleted, then for objects that have been created, and finally for relationships between single attributes as discussed above.

If an object has consistently been deleted from the Before snapshots it puts a Delete statement in the resulting script ("delete o"). If an object has been created in all examples, the engine puts a Create statement in the resulting script ("object o = createFrom prototype").

The inference algorithm deals with object creations by using prototype objects. That is, when a new object is created it is in fact copied from an existing (but often invisible) prototype object. Attributes of the newly created object can then be computed using the single-attribute inference mechanism. For example, it can be demonstrated that a newly created object should appear at the center of its layout parent.

By default, an object can serve as a prototype if it is of the same type as the newly created object, and if no more than ten attributes differ between them. If multiple prototypes qualify the algorithm chooses the one with the fewest differing attributes. If there is no such prototype for the new

1. There is actually a superior selection method for the integer inference type. This method consists of adding vectors to a new matrix one by one, making sure that every new vector is linearly independent from the ones already there. We present the above method because it can be used for all inference types, not just for integers.

object the inference engine will automatically create one. This is done by permanently copying the new object into the upper left corner of its parent and by making it invisible. It can then serve as a prototype for this object and for others.

SNAPPING

The numerical examples above all use equations that can be solved exactly. This will rarely be the case in actual demonstrations where raw solutions often read "b.x := 1.03*b.x + 0.48*b.width + 3.1" when the user intended "b.x := b.x + 1/2*b.width".

We deal with these cases by first trying if the non-constant factors can be snapped to halves and the constant factor snapped to zero (which is the case in the example above). We then try snapping the non-constant factors to halves and rounding the constant factor. If both these snapped versions of the solution fail we use the original solution.

First Try	Second Try	Third Try
$\begin{bmatrix}1.03\\0.48\\3.1\end{bmatrix} \rightarrow \begin{bmatrix}1\\0.5\\0\end{bmatrix}$	$\begin{bmatrix}1.03\\0.48\\3.1\end{bmatrix} \rightarrow \begin{bmatrix}1\\0.5\\3\end{bmatrix}$	$\begin{bmatrix}1.03\\0.48\\3.1\end{bmatrix}$

INTEGRATION WITH THE INTERFACE BUILDER

Many interface builders contain an interpreted scripting language which allows testing behavior without going through an edit-compile-link cycle. If the interface builder provides such a language the engine's inferred scripts can be made executable by translating them into the builder's native language. If the interface builder does not provide such a language one has to write a small interpreter which interprets the inference engine's scripts (so that one can interactively test the engine's inferences).

FUTURE WORK

An important shortcoming of the current engine is that it can only refer to objects by absolute name but not based on the attributes of that object. That is, it can infer a script like "obj3.width := 50; obj7.width := 50" but it cannot infer "(*.width>50).width := 50". We are currently exploring several inferencing alternatives.

Many other improvements can also be made. It is our hope that others will use and contribute to the engine.

IMPLEMENTATION

The inference engine is written in C++. It is fully implemented as described and consists of approximately 5000 lines of code. The engine is freely available for academic purposes (contact the first author).

CONCLUSION

The main difference between our inference engine and most existing demonstrational systems is that our engine does not make use of domain knowledge. The Compactor seems to be a viable alternative to using domain knowledge for reducing the amount of computation. The Inferencer can uncover linear relationships including many common ones such as aligning and centering without having domain knowledge about those special relationships. Overall, we feel that it is possible to separate the inferencing component from the presentation component of a demonstrational system, and that this separation has many benefits including increased generality and portability.

ACKNOWLEDGEMENTS

We would like to thank Siemens for their support of this research.

REFERENCES

1. Cypher, A. Eager: Programming Repetitive Tasks by Example. In *Proceedings of CHI'91* (Apr. 27-May. 2, New Orleans, LA) ACM, N.Y., 1991, pp. 33-39.

2. Cypher, A., Ed. *Watch What I Do: Programming By Demonstration*. MIT Press, Cambridge, Mass. 1993.

3. Fisher, G., Busse, D. Adding Rule-Based Reasoning to a Demonstrational Interface Builder. In *Proceedings of UIST'92* (Nov. 15-18, Monterey, Cal.) ACM, N.Y., 1992, pp. 89-97.

4. Frank, M., and Foley, J. *Inference Bear: Inferring Behavior from Before and After Snapshots*. Technical Report git-gvu-94-12, Georgia Institute of Technology, Graphics, Visualization and Usability Center, Apr. 1994. (available via: http://www.gatech.edu/gvu/gvutop.html)

5. Kühme, T., and Schneider-Hufschmidt, M. SX/Tools - An Open Design Environment for Adaptable Multimedia User Interfaces. *Computer Graphics Forum 1*, 3 (Sept. 1992), pp. 93-105.

6. Kurlander, D., and Feiner, S. Inferring Constraints from Multiple Snapshots. *ACM Transactions On Graphics 12*, 4 (Oct. 1993), pp. 277-304.

7. Maulsby, D., Witten, I., and Kittlitz, K. Metamouse: Specifying Graphical Procedures by Example. In *Proceedings of Siggraph'89* (Jul. 31-Aug. 4, Boston, Mass.) ACM, N.Y., 1989, pp. 127-136.

8. Myers, B. *Creating User Interfaces By Demonstration*. Academic Press, Boston, 1988.

9. Olsen, D., and Allan, K. Creating Interactive Techniques by Symbolically Solving Geometric Constraints. In *Proceedings of UIST'90* (Oct. 3-5, Snowbird, Utah), ACM, N.Y., 1990, pp. 102-107.

10. Singh, G., Kok, C. and Ngan, T. Druid: A System For Demonstrational Rapid User Interface Development. In *Proceedings of UIST'90* (Oct. 3-5, Snowbird, Utah), ACM, N.Y., 1990, pp. 167-177.

11. Wolber, D., and Fisher, G. A Demonstrational Technique For Developing Interfaces With Dynamically Created Objects. In *Proceedings of UIST'91* (Nov. 11-13, Hilton Head, S.C.), ACM, N.Y., 1991, pp. 221-230.