School of Informatics, University of Edinburgh

Centre for Intelligent Systems and their Applications

A Proposal for Interactive Program Generation

by

Daniel Winterstein

A Proposal for Interactive Program Generation

Daniel Winterstein

Abstract :

Programming-by-example (PBE) systems attempt to generate programs by learning a task from the user's actions. It is a field with great potential, but little success so far. Most existing PBE systems are both highly specialised and quite limited in the tasks they can accomplish. This paper sets out a new approach to PBE that is general-purpose and can handle variables, branching and loops. It could therefore offer non-experts a genuine alternative to conventional programming. Our approach makes use of automated reasoning techniques, and is based on work in interactive theorem proving using model-instance based reasoning (where general theorems are proved by considering specific cases). The 'proof-as-programs' paradigm (where theorem provers are used to generate programs) leads us to propose that model-instance based reasoning can be applied to program generation. The proposed method has the added benefit that - because of the link to an underlying logic - certain types of common bug cannot occur. We are currently working on an implementation for the domain of XML object manipulation.

Keywords : model based reasoning, program generation, macros

A PROPOSAL FOR INTERACTIVE PROGRAM GENERATION

DANIEL WINTERSTEIN

ABSTRACT. Programming-by-example (PBE) systems attempt to generate programs by learning a task from the user's actions. It is a field with great potential, but little success so far. Most existing PBE systems are both highly specialised and quite limited in the tasks they can accomplish.

This paper sets out a new approach to PBE that is general-purpose and can handle variables, branching and loops. It could therefore offer non-experts a genuine alternative to programming. Our approach makes use of automated reasoning techniques, and is based on work in interactive theorem proving using model-instance based reasoning (where general theorems are proved by considering specific cases). The 'proof-as-programs' paradigm (where theorem provers are used to generate programs) leads us to propose that model-instance based reasoning can be applied to program generation. The proposed method has the added benefit that - because of the link to an underlying logic - certain types of common bug cannot occur. We are currently working on an implementation for the domain of XML object manipulation.

1. INTRODUCTION

1.1. **The need for non-expert programming.** In the past, computer users could be divided into two groups: application users and IT professionals. However this categorisation is no longer accurate. As computing continues to expand and develop, IT skills are becoming more specialised, and 'knowing about computers' is becoming increasingly impossible. Meanwhile, as computers become an increasingly common part of everyday life, the tasks the typical user performs are becoming both more diverse and less predictable. It seems these trends can only continue; everyone is becoming a computer inexpert.

There is therefore a growing need for systems that allow non-experts to create sophisticated automations, systems that are powerful, flexible *and* easy-to-use. Many modern applications allow end-user programming (e.g. word processor macros or browsers via javascript). Such programming allows users to perform more sophisticated operations than are possible through push-button in-built procedures. Very few users though ever learn to program. They are put off by obstacles including the often steep learning curve, and the difficulty of the task. Programming in a conventional programming language requires giving the computer an abstract description of steps it is to follow. This abstraction is difficult for people in all but the simplest of situations [10].

Programming-by-example (PBE)[1] offers a solution to this problem. In PBE the user provides example actions, which the system learns, allowing it to automate the task in future. Thus PBE aims to provide the power of programming, without the difficulties. It is an area with great potential, but limited success so far.

---

[1] Also known as programming-by-Demonstration (PBD).

1.2. **Going beyond macro recorders.** Macro recorders - which used to be a common part of office applications (e.g. word-processors or spreadsheets) - provide a simple form of non-expert programming. *Recording a macro* involves hitting 'record', performing a set of actions on your document, then hitting 'stop'. Your sequence of actions can then be applied to other documents as a single action. It is a nice user friendly way of defining a macro, but it severely restricts what can be defined: there are no 'if' statements, no variables and no loops. As such, what you can achieve is quite limited. Modern office applications have by-and-large dropped the option to record macros, and replaced it with the ability to write macros in some specialised language (typically a form of Basic).

Recording a macro gives two advantages however:

(1) It is programming without having to learn a programming language, or an API (Application Programming Interface) - tasks that few users would ever consider.
(2) The user works by modifying an example case, rather than coding for an abstract case. This seems to be a more natural way of reasoning for most people [10].

This paper sets out a new PBE method - which we call *instance based programming* (IBP) - that goes beyond macro-recorders to cover variables, branching and loops. The method therefore represents a genuine alternative to programming. It makes use of theorem-proving technology, adapting a technique developed for interactive theorem proving. One consequence of this is that, by tying the program to a logic, several common types of bug are eliminated.

However PBE presents some very hard problems, and we have by no means solved them all. In particular, our proposal places high demands on the user interface, and especially on creating good clear representations. This could affect how easy the resulting system is to use, and how easy it is to extend to new domains. A series of experiments is required to gather empirical data before we can evaluate the practical value of this proposal.

In §2 we briefly discuss existing PBE systems. §3 presents a worked example of IBP. IBP is then properly defined in §4 and §5. §6 discusses HCI issues, which are especially important to PBE projects. §7 shows how IBP helps prevent programming bugs, and §8 sets out a program of future work for this project.

2. Related work

2.1. **Programming-by-Example Systems.** PBE dates back to Solomon's *Instant Turtle* system in 1971 ([12]), but it is still a relatively undeveloped field. Most existing PBE systems are highly specialised; both domain specific and limited in terms of the tasks they can accomplish. For example, Cypher's *Eager* system is very user-friendly, but it can only automate a limited range of tasks for one application making it also very restricted [2].

There are exceptions to this, including *Tinker* and *Imitate*. *Tinker* was an early PBE system capable of generating arbitrarily complex Lisp programs [7]. It is the only PBE system we are aware of that allows the user to create new data structures. Unfortunately, to use *Tinker* the user must know how to program in Lisp, how to use *Tinker*, and the details of the data-structures and functions they wish to use. It is therefore hard to see what advantages such a system offers over conventional

programming. The contrast between *Tinker* and *Eager* illustrates the trade-off between power and convenience that is, to some extent, unavoidable in PBE [9].

Imitate is a state-of-the-art PBE system which covers programs involving list, number and string manipulations. *Imitate* is capable of inferring several different types of loop from a sequence of actions, and can reason with conditions. However it has severe limitations. For generalisation to work, conditions must be entered at the 'correct' moment. Also, the learning system is intolerant of mistakes, and there is no checking for bugs in the finished program. A key limitation is that the user cannot view or edit the code generated. Empirical evidence from other PBE systems shows that this "gulf of representation" ([21]) causes various user problems [2][14]. This is a failing common to many PBE systems - and those that do allow program editing often do so only through a conventional programming language [21].

2.2. **Dr.Doodle.** Both IBP's reasoning method and its representation scheme are adapted from the Dr.Doodle system. Dr.Doodle is not a PBE system, but an interactive theorem prover developed to perform diagrammatic reasoning [19][20]. This involves proving theorems about a general case (e.g. "for all lines..."), whilst only ever working with specific cases (e.g. "consider this line here..."). Dr.Doodle does this by using models annotated with syntactic information. Proofs are constructed at the syntactic level (using a conventional forward-reasoning logic) but all variables are associated with a object instances, making up a model for the syntactic statements. The representation displays both the model and the statements in the same space. A preliminary experiment suggests that students find this a better way of working. The presence of a concrete model aids visualisation, suggests lines of reasoning and eliminates some dead-ends (c.f. *the Geometry Machine* [18]).

Dr.Doodle presents actions by showing an example before/after transformation (see figure 1). Reasoning is presented as a tree of different states, with an automatically generated natural language commentary. This allows the user to navigate the proof, and also to edit it. This representation scheme was first used in Lieberman's *Mondrian*, a drawing program which can learn new drawing tools [8].

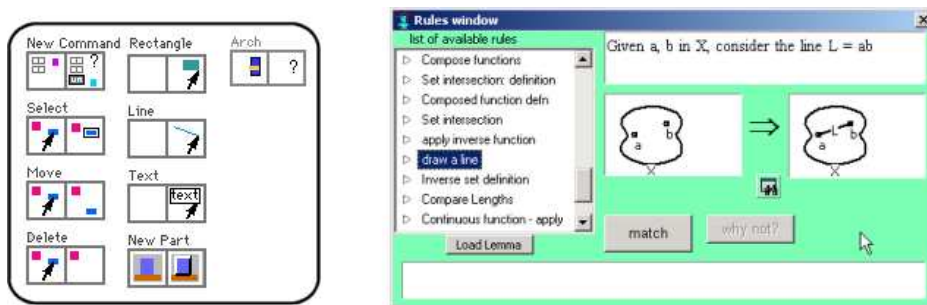FIGURE 1. Action representations in *Mondrian* and *Dr.Doodle*.

3. AN EXAMPLE: BLOCKSWORLD

3.1. **Informal overview.** This section presents a simple example of instance-based-programming in a blocksworld domain. We first give a loose overview of

how IBP works. Programs are created by recording the user's actions and their effects. This creates a chain representing the changing program state (or a tree if there is branching to handle multiple cases). The user then selects a section of the tree to be converted into a program.

Actions have both semantic effects (creating and modifying program objects) and syntactic effects (adjusting what is known about the program objects - the *observations*). A typical action will break down into three parts: pre-actions (which interactively create simple objects (e.g. positions or strings)), the main action, and implicit-observation-actions (which automatically make certain observations about the new program state). Observations are represented in a variety of ways, sometimes as explicit statements, sometimes implicitly in the object representation.

Which objects become inputs, variables and constants in the final program is determined by the section of tree chosen, and the observations produced during the recording. Conditions guaranteeing the smooth running of the program as a whole are generated by adding together the conditions for individual actions.

3.2. **Stacking blocks by example.** In this example, we will use three actions: `input-position`, `move-object` and `observe-X-on-Y`, shown in figure 2. Actions are represented by 'dominoes', where the left-hand diagram shows an example input state and the right-hand diagram shows the action's effects.
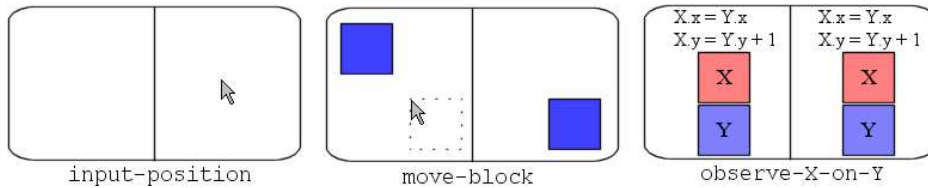

FIGURE 2. A pre-action, the move action and an implicit-observation-action.

Figure 3 shows how the program tree develops.

(1) Starting in program state ① the user selects the `move-block` action.
(2) The system attempts to match the `move-block` input state with the current program state. `move-block` has two inputs (a block and a position) and one condition (that the target position should be empty - this is represented via a graphical convention). The system asks the user to select a block (the user selects 'A'), and triggers an `input-position` pre-action to create a position from a user click. It then checks that this gives a valid match between the action's input state and the current program state.
(3) The matching is valid, so `move-action` is applied, creating new program state ②
(4) The system checks for any implicit-observation-actions that match the new state. It finds and applies `observe-X-on-Y`.
    Although the `observe-X-on-Y` action looks like it should do nothing, it does have effects. This is because its input conditions (A.x=B.x, A.y=B.y+1) are true in ②, but have not been observed (that is, they are not known to be true); `observe-A-on-B` observes them. When applying `observe-X-on-Y` the system first tries to prove the conditions from existing observations. This fails, so a case split is introduced producing program states ③ (with

the new observations) and ④ (a new automatically generated case where the conditions are false).

(5) The user is not interested in the case "A not on B", and kills this branch (program state ⑤).

Suppose the user now chooses to produce a program from this example. The system determines that the resulting program should have 2 inputs (blocks A and B) with the condition "nothing on B". The new program is represented by a domino created from the program tree (figure 4), and is given a name by the user. We will call this action 'stack'.
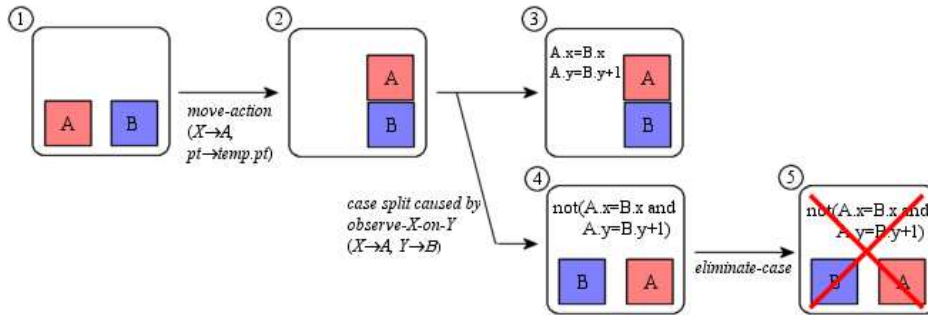

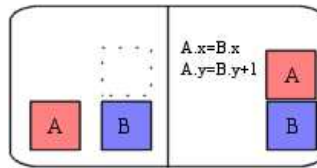FIGURE 3. A program tree for placing one block on another.

FIGURE 4. Domino for new `stack` action.

At this point, the program we have recorded will take any block and put it on any other block. We want it to stack the blocks in order, so we specify the relationship between A and B:

(6) The user adds the observation "B=A+1". This step can be done at any point. It creates a case split ("B=A+1" or "B≠A+1").

(7) The user kills the "B≠A+1" branch.

Now when generalising, we will get a program 'stack' that given two blocks, places them in order. Creating a program that will stack an arbitrary number of blocks is slightly more complicated. Too save space and present a simpler picture, we will allow blocks to 'float' (i.e. we will create a sort routine - which shows that we could create a stack routine). First we define an 'unstack' action that unstacks two out-of-order blocks. Given this, and a meta-action 'repeat' (c.f. §4.3), we build a program for sorting a collection of blocks as shown in figure 5. Given a list of blocks (with (...) representing a list), the program applies the unstack action as many times as possible, followed by the stack action as many times as possible. This will terminate when the blocks are stacked in order.
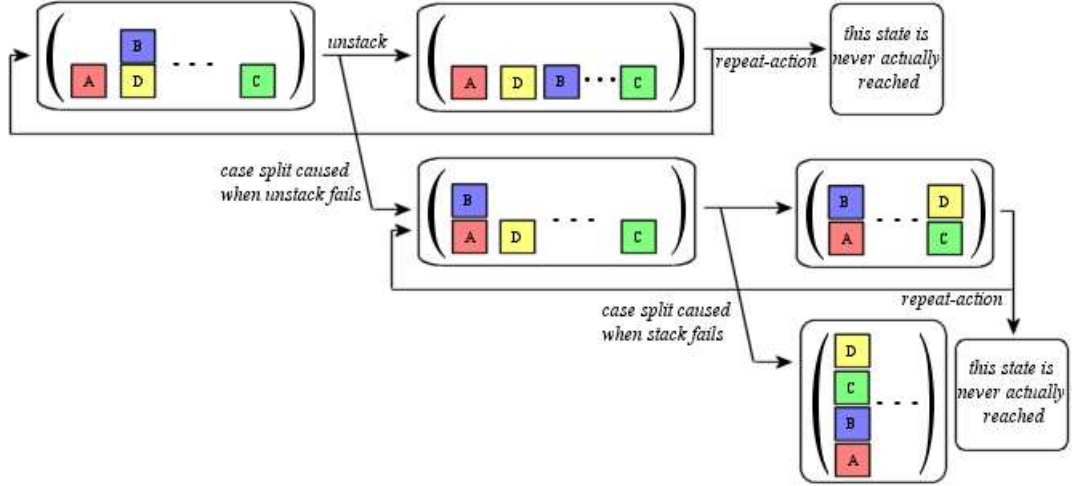
FIGURE 5. The full stack blocks program.

4. INSTANCE BASED PROGRAMMING

4.1. **IBP terms.** We first define some terms, then talk about how they work in §4.2.

Definition 4.1. *Objects* are logical entities with properties (e.g. X, $X.type = number$, $X.value = 5$). Property values are also objects. There are a set of basic objects, from which all others are derived. Technically, an object's properties cannot change, and changes are effected by replacing the object with a similar one.

Definition 4.2. *Observations* are logical statements about objects (e.g. $\texttt{positive}(X)$, $X < Y$)

It is important to note that we distinguish between observations (which are 'known' facts, and important to the program) and object properties (which are not known, and irrelevant to the program). In the course of creating a program, object properties can be observed. This distinction, and the ability to add new observations, will be important in solving the generalisation problem.

Definition 4.3. A *program state* consists of a set of objects and observations, ordered according to the order in which they were created. Given a program state P we will write $objects(P)$ and $observations(P)$ for the objects and observations of P respectively.

Definition 4.4. A *matching* maps objects in one program state with those of another program state, and also specifies some generalisation information. Formally, a matching m is a function from $objects(P) \rightarrow objects(P') \times \{any, all\}$. $m(X) = (Y, any)$ denotes that X has been matched to Y here, but any valid matching could be used. By contrast $m(X) = (Y, ask)$ means that when replaying this action, the system should ask the user to choose between good matchings. This mechanism provides a minimally intrusive way of specifying user input in the final program.

Definition 4.5. A *program* is a directed graph of program states, with arcs labelled by action/matching pairs.

Definition 4.6. An *action* creates new program states by creating and/or modifying objects and observations (e.g. "Given X such that X.type = number, create Y, Y.type = X.type, Y.value = X.value + 1").

An action is itself a program (i.e. it is a directed labelled graph of simpler actions). The initial state represents the action's pre-conditions - the objects that must exist, and the observations that need to be proved before the action can be used. The final states (i.e. the leaves of the program-tree) represent the action's effects.

Definition 4.7. The *inputs* for an action are the objects in the actions initial state.

Definition 4.8. The *conditions* for an action are the observations in the action's initial state.

Definition 4.9. An *observe-action* is a particularly simple action that makes observations about the program objects without changing them.

Definition 4.10. An *instance-based programming system* consists of:
- A first-order quantifier-free language with equality for stating object properties and observations.
- A logic (i.e. a set of axioms and inference rules) for reasoning about observations
- A set of basic actions (e.g. creating objects, modifying properties, asking for input)
- A set of observe-actions to be automatically applied. We call this set the *implicit observation actions*. Their purpose is illustrated in §3 by the `observe-X-on-Y` action, and explained in §6.1.

These definitions are too general for an actual system: we have not specified what the basic objects and actions are, nor what the language for stating observations is, or the logic for proving them. This is deliberate, since the method we describe will work regardless of these details.

4.2. **Creating a program tree.** Starting from an initial empty state (no objects, no observations) the user creates a program tree by applying actions; the program tree is a structured recording of those actions.

To apply an action, the user must first create a matching between the objects in the action's initial state and objects in the current program state. Typically, creating a match should be a painless process that can be conducted with a simple point-and-click interface (e.g. 'delete object' requires only one object to be specified to give a matching). There are some subtleties to this however, which are discussed in §6.

When trying to apply an action, there are three possibilities:
(1) The conditions follow from the observations.
   ⇒ apply the action, creating a new program state as the child of the current program state.
(2) The conditions do not follow from the observations, but are true (i.e. the conditions follow from the properties of the program state).
   ⇒ perform a case split:
     - Case A: the conditions are true
       Create a new program state, add the conditions as new observations and apply the action.

- Case B: the conditions are false
  Create a new program state, add the negation of the conditions as new observations and modify the program state to give a state where the conditions are indeed false.
(3) The conditions do not follow from the observations and are false in the example.
  ⇒ the action cannot be applied. If we are in a loop (c.f. §4.3 below), introduce a case split, creating a branch 'action X cannot be applied' leading to a copy of the current program state. Otherwise do nothing.

After applying an action, we check the implicit observation rules, and apply any of those that match.

4.2.1. *Tractability.* The procedure given in §4.2 requires automated reasoning (AR) technology for several steps. We need to consider whether these steps are tractable under 'normal' conditions (and what can be done if they aren't). AR is used for:

- Testing whether an action's conditions follow from the current state's observations.
  Note that we are working within a quantifier free first-order logic with equality, which is decidable [13]. Moreover, there are a number of fast efficient theorem provers available for solving such problems [17]. Should a test be too slow, we can proceed by assuming that it fails - if wrong, this will only introduce an unnecessary case split.
- Testing whether an action's conditions are true in the current state (i.e. that the conditions are consistent with the properties of the program objects). This test is also a first-order quantifier-free problem, and it is highly constrained. It should therefore be fast. Should a test be too slow, we can proceed by assuming it succeeds and perform a case split. We would no longer know for sure that the action can be applied (although note that the user believes it can be).
- Generating a modified example (case 2 above). This is also decidable. Such problems are less tractable but efficient systems are available for tackling them (e.g. *MACE* or *Gandalf* [17]). Note that we want to find not just a valid example, but a *good* example (i.e. one that doesn't throw the user). There is a simple heuristic which should both improve speed, and give good examples. In most cases, only a small change will be required to make the current example fit the new conditions. Hence it will often be possible to generate the new example by gradually modifying the current one. As the layout modification algorithm used in *DiaGen* demonstrates ([6]), modification problems can be fast even when the corresponding generation problem isn't. Should a model-modification step be too slow, we can ask the user to perform the modification. If they fail, we can proceed by using the unmodified model annotated with the modified conditions. This introduces a discrepancy between the observations and the object properties. This discrepancy could be confusing and lead to errors, whereby an action should be applicable according to the observations.

We envisage that when demonstrating a program with IBP, users will typically work with quite simple examples that don't involve simultaneously manipulating large numbers of objects. We therefore anticipate that the tests will be quick. The

model-generation problem could still cause problems. However as explained above, we can proceed even when the AR routines are slow, albeit at the cost of potential bugs.

4.3. **Meta-actions.** We need four special meta actions that modify the program tree:

(1) *Eliminate-case*: Remove one branch of a case split, and add the negation of it's observations to all states (this will then give an extra condition on when the final program can be used).

   We cannot make assertions about objects until they (or an object known to be equal) has been created, so this can involve 'pruning' the program: If eliminating a case imposes observations on an object that isn't present (modulo equality) in the initial program state, we must throw away those states that don't contain the object.

(2) *Observe*: Make an arbitrary observation. The user needs to learn the observation language to use this action. However this should be easy compared to learning a programming language.

(3) *Repeat*: Select an earlier program state, and introduce a case split:
   • Case A: The current state does not have a valid match with the earlier program state.
     ⇒ Create a new program state identical to the current one.
   • Case B: The current state matches the earlier state.
     ⇒ Jump back to the earlier state and attempt to replay the actions recorded from there (c.f. the behaviour for when an action fails in §4.2).

   The user may want to think about a loop before starting it, and should be given the option to browse the program tree before initiating looping behaviour. When replaying a loop, the steps should initially be performed slowly, then speed up, with the user able to stop the loop at any time. If they do so, the program reverts to the state it was in before `repeat` was applied.

   Note that from this basic loop action, other types of looping behaviour can be constructed (e.g. *for* loops by introducing a counter).

(4) *Create-new-action*: Create a new action from a program. This is a complicated step, described in §5.

5. GENERALISATION

Generalisation is the biggest problem in PBE. The difficulty is that extra information is needed to turn an example into a program (typically, we need to know what the user *meant* by a given action [16]). Some systems get this information by asking the user (e.g. *Peridot* [15]). This makes them 'clunkier' to use, and does not scale well [10]. Others use heuristics to guess the extra information (e.g. *Mondrian*). Typically this makes the system inflexible and hard to extend. Some systems simply restrict the system's range (e.g. *Eager* [2]). Ultimately there is no optimal solution [9].

However, we believe our method might deliver a reasonable amount of ease-of-uses without compromising range. The real work of generalisation is carried out whilst editing the example by the twin processes of adding observations and pruning

the program tree. The generalisation procedure then creates the most general program possible given the observations. The user can further restrict the program by adding extra conditions. In fact, since the new action produced by generalisation keeps the original program tree, it is possible to view and edit programs using the same interface as for recording the original example. This editing ability is one of IBP's advantages over most existing PBE systems, where the original example is thrown away [21].

5.1. **Simple generalisation.** To create a program from the example all that is required is that the user select a section of the program tree. The first state then become the input state for the new action, and those objects that exist in the first state are the program inputs. The final state(s) mark where the program stops. The new program is executed (when used with a valid matching) by replaying the recording. For example, in the stack block example (figure 3), the inputs are A, B and c, with the conditions $type(A) = block, type(B) = block, type(c) = position, c.x = B.x, c.y = B.y - 1$.

5.2. **Improved generalisation.** In the stack-block example of §3, it is clear that we can calculate where block A should be moved to from the post-conditions: A.x=B.x, A.y=B.y+1. Therefore the input position is fixed, and does not need to be an input. We want to take advantage of this, by converting program inputs into functions of other inputs wherever possible.

This is a computationally difficult problem in general. We can determine that an input X is unnecessary by proving $conditions(initial - state \backslash \{X\}) \vdash X$. Thus we can calculate a maximal set of redundant inputs (though there is not necessarily a unique solution). Calculating such inputs when the action is applied is a model-generation problem - potentially intractable. We cannot rely on such operations to be fast enough, especially for steps that might be repeated many times in a program. We therefore only detect and remove redundant inputs for which we can find arithmetic functions (e.g. $x = y + z$). Finding such functions is a simple matter of solving linear equations. Calculating them when the action is applied is, of course, very fast. There can be multiple maximal sets of redundant inputs, in which case any one can be used; the resulting programs will perform the same operations. The user may have preferences as to which inputs are specified and which are calculated, so they should be consulted in such cases.

6. User interface design

Program generation in PBE is always done through a development environment, and a good development environment is crucial to any useful system. The IBP development environment needs to provide the following:

- Facilities for inspecting program states (i.e. representations for objects and observations)
- A view of the program tree
- A working space (displaying the current program state)
- A user-friendly system to apply actions
- An easily searched store of basic objects
- An easily searched store of basic actions

6.1. **Representation.** The key difficulty here is probably producing good representations - a non-trivial HCI problem. The representation scheme has to cover objects, observations and actions and should be easy to learn, if not intuitive. Inevitably there will be trade offs in creating a general purpose representation. However as UML demonstrates, reasonable general-purpose representations are possible. Moreover the representation can be interactive. This makes the task much easier, since not all the details of a program state have to be displayed at once.

Some objects, such as text documents or bitmaps, have an obvious representations and can be presented 'as themselves'. Other objects, such as databases or networks, require a symbolic representation. Then there are objects such as 3D models, for which many views are possible, and which is best can depend on the context.

For representing observations, we adopt the scheme used in *Dr.Doodle*. Observations can be represented using several modalities: as logic statements, as graphical annotations (e.g. highlighting) or implicitly, as part of an object's representation. Logic statements are the easiest to implement, simply being added as annotations, either near the relevant objects, or in a list to one side. Graphical annotations can be much more intuitive, but a representation has to be built into the system for each such observation. Implicit observations are used for relations that 'stand out' as being clearly important from the object representation. They therefore do not need to be explicitly stated - in fact, such relations are only stated if they are either not observed (e.g. "irrelevant(A.x=B.x)") or negated. Examples might be object type, or "file X in folder Y". These relations are observed by default, using the implicit-observation-rules[2]. If they are not relevant (i.e. the program should generalise over them), then this has to be specified by the user. Using objects representations to convey certain statements in this way, allows us to use simple representations where possible, without sacrificing the ability to represent complex relations.

Given good representations for objects and observations, actions can be represented by an example before/after transformation. This is the representation used in both *Mondrian* and *Dr.Doodle* (c.f. figure 1). Programs are represented as browsable trees of the different program states. An accompanying natural-language description of the program can also be given (generated by associating text templates with each action - c.f. both *Dr.Doodle* and *Mondrian*).

6.2. **Applying actions.** To apply an action, the user selects an ordered set of objects to act as inputs. This sounds simple, but in practice it could well be irritating. For example, when using the `move-block` action in figure 2, most users would expect to select the target block, then click on the desired destination. However `move-block` requires the destination to be given as an input. Thus instead the user must first create a position object at the destination, then select it and the block.

We introduce *pre-actions* to allow the natural way of applying actions to work. A pre-action creates an object, asking for user input in an appropriate way to specify object properties (e.g. a mouse click for a position, a text prompt for a string, or a dialog form for more complex objects). When the user selects an action such as *move-block*, the system identifies the inputs for which it has matching pre-actions. The other inputs are entered first, by selecting matching objects in the program

---

[2]Thus in figure 4, we could hide the A.x=B.x, A.y=B.y+1 post-conditions since they are clearly suggested by the object representations.

state. Then the pre-actions are applied, completing the set of action inputs. The order in which this is carried out is determined by the order in which the objects were originally created. As with the selection of existing objects, these input objects can be selected specifically ('ask') or arbitrarily (labelled 'any'). When the program is replayed, 'any' objects will be automatically created (a model generation task, but almost certainly a simple one), whilst 'ask' objects will be created from user input.

7. Reducing program bugs

By attaching necessary conditions to each action, and using these to calculate conditions, instance-based programming makes certain bugs impossible:

- Bugs caused by un-initialised variables or referencing the wrong variable are eliminated because the user handles concrete object instances rather than variables.
- More importantly, hidden assumptions are always exposed, appearing as explicit conditions on the final program. This should prevent a wide variety of errors.

The IBP method given above does not reason about the effects of loops. For example, with the stack blocks program presented in figure 5, IBP does not prove that this will (a) stack the blocks in alphabetical order, or (b) terminate at all. Another potential source of bugs is when a program runs smoothly, but has unwanted effects. However, since there is always at least one example where the program must behave as desired (that is, the example from which the program was generated), such bugs should be less common than with conventional programming.

8. Future Work

We are currently working on a system implementing the ideas set out in this paper. The end goal of such research is to produce user-friendly systems for use by non-experts. It is therefore important to show not just that an idea works (i.e. can be implemented), but also that it could meet this goal. To do this, we intend to compare the performance of non-programmers using an IBP system versus an equivalent programming system. These experiments will be conducted in the domain of manipulating (simple) XML objects. XML forms the backbone for many web-related technologies, making it an important area for non-expert programming. This domain also has the advantage that XML objects are often closely linked to a representation (via HTML/XSL). It is thus an attractive area for applying PBE.

If these evaluations are successful, we will develop the capabilities of our current system to cover a more interesting range of objects and actions (i.e. by adding more actions and specialised object and observation representations). The ease or difficulty involved in doing so will be another important test of this approach.

We will also explore what IBP cannot do, and look at extending IBP to cover programming techniques such as *methods* (i.e. actions associated with objects), *recursion* (perhaps best done by the merging of actions: given an action for the base case and an action for the step case, merge to produce a recursive program). Another extension would be to allow the user to view and edit the underlying logic used for proving conditions. Assumptions (e.g. "X a date $\Rightarrow$ X=dd.mm.yy") could be added to the logic. Since observe-actions can be converted into logical assumptions, this would be a simple extension. This would make the system smoother to

use, since more conditions would be provable and therefore fewer case splits would be generated. Using dependency tracking[3], the system can keep track of which assumptions are needed in creating a program. These assumptions can then be added to the final program as extra conditions; normally hidden, but inspectable if required.

9. Conclusion

In this paper we have set out a new method - instance-based programming (IBP) - for generating programs. Instance based programming involves performing general-purpose computation, whilst only ever working with specific examples. The correct generalisation is then extracted from observations made whilst working with the example case. IBP is based on an interactive theorem proving technique, and uses AR technology - theorem-proving and model-generation. It potentially has considerable benefits over using programming languages. The user has no need to learn a programming language, and can work with concrete examples rather than having to think in abstract logical terms. These features make IBP a possible solution to non-expert programming. Also, IBP prevents some bugs and discourages others - a feature that will be attractive to all computer users. Whether or not IBP can really deliver these benefits though, has yet to be proven. The next stage of this project is to implement a prototype system for emperical testing.

References

[1] A.Cypher, ed. "Watch What I Do: Programming by Demonstration" MIT Press, 1993.

[2] A.Cypher "Eager: Learning Repetitive Tasks by Demonstration" in [Cypher, et. al.], 1993.

[3] E.Denney "Logic-based Program Synthesis via Program Extraction" AAAI 2002 Spring Symposium, 2002.

[4] E.Furse "Imitation: a Solution to End-User Programming" in proceedings *First International Conference on End User Programming*, 2001.

[5] J.Kalman "Automated Reasoning with Otter" Rinton Press, 2001.

[6] O.Köth & M.Minas "Structure, abstraction and direct manipulation in diagram editors" in *Diagrammatic Representation and Inference*, Springer-Verlag, 2002.

[7] H.Lieberman "Tinker: A Programming by Demonstration System for Beginning Programmers" in [Cypher, et. al.], 1993.

[8] H.Lieberman "Mondrian: A Teachable Graphical Editor" in [Cypher, et. al.], 1993.

[9] H.Lieberman & D.Maulsby "Instructible agents: Software that just keeps getting better" IBM Systems Journal, 35(3&4), 1996.

[10] H.Lieberman "Art Imitates Life: Programming by Example as an Imitation Game" in *Imitation in Nautural and Artificial Systems*, MIT Press, 2002.

[11] L.Paulson "The foundation of a generic theorem prover" Journal of Automated Reasoning, 1989.

[12] D.Maulsby & A.Turransky "A Programming by Demonstration Chronology: 23 Years of Examples" in [Cypher, et. al.], 1993.

[13] D.McAllester "Lecture Notes for Artificial Itelligence: First Order Logic" MIT, 1992.

[14] R.McDaniel & B.Myers "Gamut: Creating Complete Applications Using Only Programming-by-Demonstration", unpublished, available online at http://www.cs.cmu.edu/∼amulet/papers/gamuttochi.ps, 2000.

[15] B.Myers "Creating interaction techniques by demonstartion" in *Visual Programming Environments: Paradigms and Systems"*, IEEE, 1990.

[16] P.Piernot & M.Yvon "The AIDE Project: An Application-Independent Demonstrational Environment" in [Cypher, et. al.], 1993.

---

[3]*Dependency tracking* returns the axioms and assumptions used to prove a statement. It can be implemented with many theorem provers (e.g. *Otter* or *Isabelle* [5][11].)

[17] T.Tammet "Towards Efficient Subsumption" CADE 98 pp.427-441, Springer Verlag, 1998.

[18] D.Wang Geometry Machines: From AI to SMC proceedings of the *3rd International Conference on Artificial Intelligence and Symbolic Mathematical Computation*, LNCS, 1996.

[19] D.Winterstein, A.Bundy & M.Jamnik "A Proposal for Automating Diagrammatic Reasoning in Continuous Domains" in *Theory and Application of Diagrams*, Springer-Verlag, 2000.

[20] D.Winterstein, A.Bundy, C.Gurr & M.Jamnik "Using Animation in Diagrammatic Theorem Proving" in *Diagrammatic Representation and Inference*, Springer-Verlag, 2002.

[21] T.Wright & A.Cockburn "A Language and Task-based Taxonomy of Programming Environments" to appear in *2003 Symposium on Visual Languages and Formal Methods*, IEEE, 2003.

E-mail address: `danielw@dai.ed.ac.uk`