Similarity Inheritance: A New Model of Inheritance for Spreadsheet VPLs

Rebecca Walpole Djang and Margaret M. Burnett
Department of Computer Science
Oregon State University, Corvallis, Oregon 97331
{walpolr, burnett}@cs.orst.edu

Abstract

Although spreadsheets can be argued to be the most widely-used visual programming languages (VPLs) today, most are very limited compared to other VPLs, supporting only a few built-in types and offering only primitive support for code reuse. The inheritance mechanisms of object-oriented programming might seem to offer help for the latter problem, but incorporating these mechanisms in a traditional way would introduce concepts foreign to spreadsheets, such as message passing. In this paper, we present similarity inheritance*, a new approach to inheritance that is suitable for seamless integration into the spreadsheet paradigm. We first explain the model independently of any implementation, and then present a prototype implementation in the research spreadsheet VPL Forms/3. We show that bringing inheritance functionality to the spreadsheet paradigm can be done using the widely-understood idea of copy/paste. Further, we show why the approach requires the presence of a live, visual environment.*

1. Introduction

Spreadsheets have proven to be a popular programming paradigm, accessible even to non-programmers. Current spreadsheets, however, suffer from some of the problems that have been solved in other programming languages. For example, in other programming languages, object-oriented inheritance mechanisms have improved upon ad-hoc (cut-and-paste) reuse of code, but spreadsheets still support only ad-hoc reuse through copy/paste and formula replication. Thus spreadsheet users must remember the reuse relationships themselves and maintain them manually whenever a reused formula changes. Some commercial spreadsheets such as Excel® have a few additional conveniences, such as automated formula adjustment when a new copy of a linked spreadsheet is made. However, these features are simply editing conveniences, and the user is still left to manually maintain the reuse relationships.

It occurred to us that incorporating inheritance into spreadsheets could result in stronger support for formula reuse than is found in current spreadsheets. However, existing models of inheritance do not seem suitable for the spreadsheet paradigm because they introduce concepts foreign to spreadsheets, such as message passing. Thus, we set out to find an approach to inheritance suitable for spreadsheet VPLs.

We use the term *spreadsheet VPLs* to refer to a variety of systems that follow the spreadsheet paradigm, from commercial spreadsheets to more sophisticated research VPLs that follow the declarative, one-way constraint evaluation model. The essence of the paradigm is summarized by Alan Kay's *value rule* for spreadsheets [9], which states that a cell's value is defined solely by the declarative formula explicitly given it by the user.

In this paper we present a new approach to inheritance suitable for spreadsheet VPLs, and an instantiation of the approach in the research spreadsheet VPL Forms/3 [1, 4]. The approach, called *similarity inheritance*, provides a concrete way of sharing behavior among objects in a spreadsheet VPL. The unique attributes of similarity inheritance are that:

- it includes an explicit visual representation of all the object's unique and shared behaviors, rather than leaving some behaviors implied through parenthood;
- it is flexible enough to allow sharing at multiple granularities and even allows mutual inheritance;
- it brings object-oriented concepts to spreadsheet VPLs without using external languages or macros;
- it subsumes the current spreadsheet edit-based mechanisms for formula propagation, unifying formula reuse with inheritance.

2. Related work

2.1 Alternatives to traditional inheritance

Our approach aims at providing more concreteness than is found in the traditional class/subclass model. This goal is also characteristic of the prototype model of inheritance [11], which is based on concrete parent objects rather than abstract classes. In the prototype model, if an object cannot handle a message directly, it delegates it to its parent object, which in turn handles it or delegates it to *its* parent, and so on. Prototypes remove the need for the concepts of class, subclass and instance since any object can be used as the basis for defining a new object. Self [20] is perhaps the best-known prototype-based language. ObjectWorld [15] is a prototype-based language that uses visual mechanisms to emphasize concreteness.

In the prototype model, objects include their definitions, whereas in our approach definitions include objects. Another important difference between our similarity inheritance approach and most prototype-based languages is that our model does not use message passing. Kevo [18] is one of the few prototype-based languages that does not use message passing. Kevo is a textual language that emphasizes the concreteness and self-sufficiency of objects. Operations can be marked as applying to individual objects or to *clone families* which are groups of similar objects automatically inferred by the system. Thus Kevo does not require a designated parent prototype for a collection of objects, but consequently there are no change propagation mechanisms for objects outside the clone family.

Most approaches to inheritance operate at the granularity of entire classes or objects. Mixins [2] are a technique for providing inheritance of partial classes. Also known as abstract subclasses, mixins exist only to be inherited by other, complete classes. They usually define just a small piece of functionality and, combined with multiple inheritance, can cut down on the code duplication that arises when the language allows inheritance only at the class level. Another approach to fine-grained inheritance is found in the language I^+ [14]. I^+ inheritance is not determined by subclassing, but by explicitly listing the methods to inherit. Neither of these approaches supports a granularity finer than whole methods.

2.2 Combining spreadsheets with object-oriented programming

Many VPLs have incorporated object-oriented concepts [3, 5, 6]. However, spreadsheets and more advanced spreadsheet VPLs have had little work to date on approaches to inheritance, perhaps because there has been only a little work that incorporates support for objects into spreadsheets.

Commercial spreadsheets provide support only for a few built-in types—numbers, Booleans, and strings—as first-class values,and do not provide a formula-based mechanism allowing users to add new types of objects. Although some spreadsheets gain partial support for additional objects through the use of macro languages and “trapdoors” to other programming languages (such as Visual Basic), these approaches do not maintain a seamless integration with the spreadsheet paradigm, because they use notions such as global variables, state modification, and imperative commands in a language different from the formula language of the spreadsheet.

A few research spreadsheet VPLs have also incorporated external languages to support object-oriented features. ASP (Analytic Spreadsheet Package) is a spreadsheet VPL in which every cell can be any object, and every formula is written in Smalltalk [16]. Smedley, Cox, and Byrne have incorporated the VPL Prograph with user interface objects into a conventional spreadsheet for GUI programming [17]. C32 [12] is a spreadsheet VPL that is part of the Garnet user interface development environment [13]. C32 does not itself feature the graphical creation and manipulation of objects; these must be created by other tools or written in Lisp. These approaches add some of the power of object-oriented programming, but do not enforce consistency with the value rule, since global variables and state-modifying mechanisms circumvent it.

Some research spreadsheet VPLs have moved toward expanding the types of objects supported without the use of external programming languages. A pioneering system in this direction was NoPumpG [10] and its successor NoPumpII [22], two spreadsheet VPLs designed to support interactive graphics. These languages include some built-in graphical types that may be instantiated using cells and formulas, and support limited (built-in) manipulations for these objects, but do not support user-defined objects.

Penguims [8] is a spreadsheet VPL for specifying user interfaces. Penguims supports composition of objects by collecting cells together, and formula inheritance at the object level. Unlike our work, it employs several techniques that do not conform to the spreadsheet value rule, such as interactor objects that can modify the formulas of other cells, and imperative code similar to macros.

3. Background: Introduction to Forms/3

We have created a prototype implementation of our approach to inheritance in the spreadsheet VPL Forms/3, and the examples in this paper are presented in that language. This section provides the background in Forms/3 necessary to understand the examples.

A Forms/3 programmer creates a program by using direct manipulation to place cells on forms (spreadsheets) and to define a formula for each cell using a flexible combination of pointing, typing, and gesturing. A program's calculations are entirely determined by these formulas.

Forms/3 has long supported an extensible collection of types. Attributes of a type are defined by formulas in cells, and an instance of a type is the value of a cell, which can be referenced just like any cell. For example, the built-in circle object shown in Figure 1 is defined by cells defining its radius, line thickness, color, and other attributes. One way to instantiate a circle is to copy the circle form, changing any formulas necessary to achieve the desired attributes (as in the figure); another way is to graphically define its attributes [4], such as by sketching a new circle or by stretching an existing circle by direct manipulation. The graphical method is a shortcut for the first method, and we will use only the first method in this paper.

To implement a new user-defined type of object, the Forms/3 programmer provides cells and formulas to construct a prototypical object. During the construction the system responds, as one would expect on a spreadsheet, with immediate visual feedback as each new formula is entered. The formulas specify the internal composition of the object, how it should appear visually on the screen, and any operations that it provides.

The internal composition is defined by cells and matrices that can be placed inside *abstraction boxes*, which provide information hiding. Abstraction boxes are cells whose formulas default to being the composition of their components. For example, Figure 2 shows a stack implemented by a one-dimensional matrix (inside

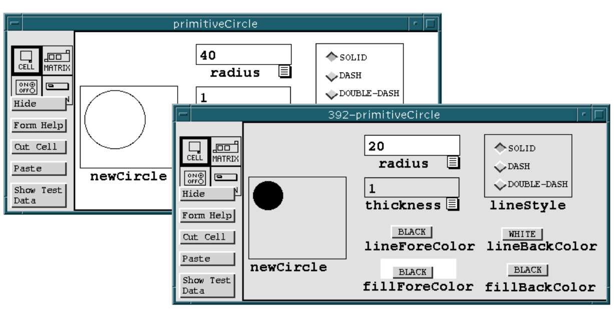

Figure 1: The white *primitiveCircle* form is a built-in form that defines a prototypical instance of type primitiveCircle. The gray *392-primitiveCircle* form is a copy that has been modified to describe a different instance. The circle in cell *newCircle* is defined by the other cells, which specify its attributes. To refer to the circle elsewhere in the program, a formula can reference *392-primitiveCircle:newCircle.* The programmer cannot view the formula (primitive implementation) of *newCircle*, but can view and specify the other cells' formulas by clicking on their formula tabs ( ). Radio buttons and popup menus (e.g., *lineForeColor*) provide a way to reliably enter constant formulas when only a limited set of constants are valid.

abstraction box *Stack*), in which the programmer has placed the sample value "hi". Because cell *Stack* has a sample value, as soon as the formula for cell *top* is entered, *top* displays *Stack*'s top element ("hi"). The other formulas are also programmed in this concrete way; they reference *Stack* and immediately display their own results based on the sample. The sample values (in cells with visible formula tabs) on copies of form *Stack* can be replaced by references to other cells in the program, which provides the functionality of incoming parameters in traditional languages.

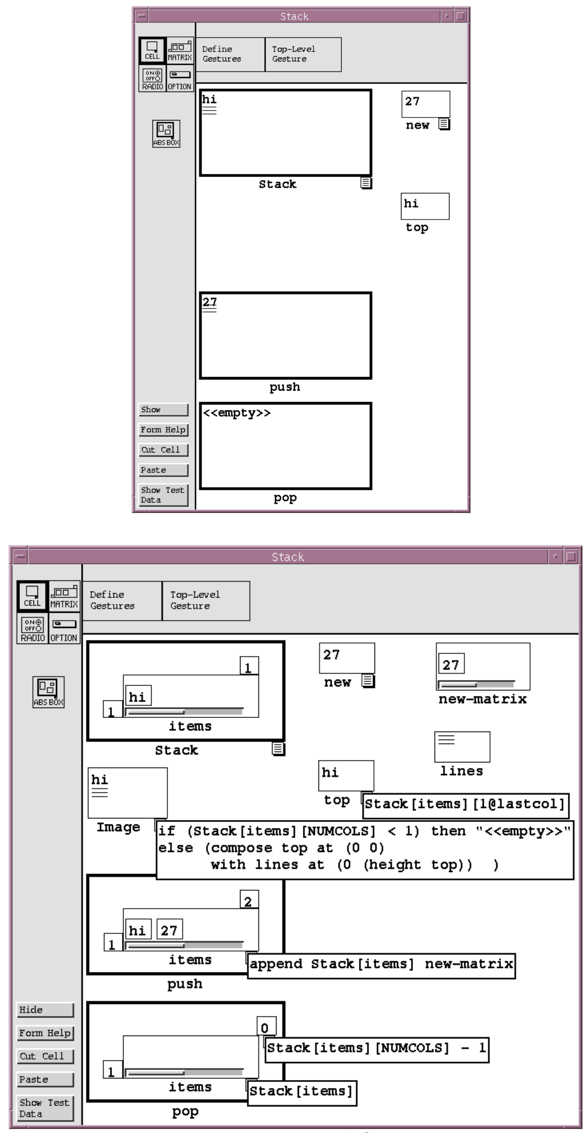

Figure 2: (Top): The user's view of Stack hides the internal implementation and displays stacks using the formula the programmer has provided for a distinguished *Image* cell. (Bottom): The stack implementor's view of object Stack with most of the cell formulas visible. (Matrices in Forms/3 are not required to be homogeneous.)

4. Similarity inheritance model

In this section we show how the preceding approach to objects in spreadsheets can be extended to support inheritance. We begin by describing our new model of inheritance, independently of any language implementation. In the model description, we will use object-oriented terminology to facilitate comparison with other models of inheritance, although it will later be demonstrated (Section 5.5) that the approach is not restricted to relationships among objects, and can be used for relationships among Excel-like spreadsheets as well.

We define the similarity inheritance model to be comprised of a model of interaction (between the programmer and the computer) and a semantic model. The interaction model is composed of the tuple (`C,F,L,R`) where `C` is the copy operation that creates a shared definition, `F` is the formula definition operation, `L` is a liveness level 3 or higher from Tanimoto's liveness scale [19] indicating that immediate semantic feedback is automatically provided[1], and `R` is a representation mechanism that explicitly includes all shared formulas and relationships in each object's definition.

Three important points about the interaction model are: (1) it separates the syntax with which the human communicates to the computer about program semantics from that used by the computer to communicate to the human about program semantics (for example allowing animations or graphical views), (2) it does not necessarily map to a static textual syntax (for example, it allows dynamic syntaxes) and (3) it depends on environmental characteristics that are not usually guaranteed by textual programming languages but are common in visual ones. Note that the elements of the interaction model are not mere editing details of an environment, but rather define the general characteristics upon which our semantics rest.

The semantic model can now be defined as follows. Each object definition D in a program is a set of behaviors $\{D_{b1}, D_{b2}, \ldots, D_{bn}\}$. For example, in Forms/3, an object definition is a form. Each D_{bi} is a formula residing in a cell. The symbol $\rightarrow$ (pronounced "shares with") indicates a shared behavior; the arrow points from the original version to the copied one. The semantics of $\rightarrow$ are

$A_{bi} \rightarrow B_{bi} \Rightarrow A_{bi} = B_{bi}$.

[1]At liveness level 1 no semantic feedback is available. At level 2 the user can obtain semantic feedback, but it is not provided automatically (as in interpreters). At level 3, incremental semantic feedback is automatically provided after each program edit, and all affected on-screen values are automatically redisplayed (as in the automatic recalculation feature of spreadsheets). At level 4, the system responds to edits as in level 3, as well as to other events such as system clock ticks.

The operations `C` and `F` determine when $\rightarrow$ holds, as summarized in the following table ($i \in \{1..n\}$ and the table omits postconditions that duplicate preconditions):

Operation	Precondition	Postcondition
`C` applied to A	A is an object definition B does not exist	B exists and $\forall i, A_{bi} \rightarrow B_{bi}$
`C` applied to A_{bk} and B	A and B are object definitions and $A \neq B$	$A_{bk} \rightarrow B_{bk}$
`F` applied to B_{bk}	B is an object definition	$\forall A, A_{bk} \not\rightarrow B_{bk}$

The first row defines large-grained similarity, and means that if `C` is applied to an object definition A, a similarity relationship will be created between A and a new object definition B such that all of A's behaviors share with B (this can be abbreviated $A \rightarrow B$). The second row defines fine-grained similarity, which allows a single behavior A_{bi} to be copied to object definition B to create a shared relationship between A_{bi} and B_{bi}. The third row implies that overriding removes "upstream" sharing relationships, but not "downstream" relationships.

Due to element `R` of the interaction model, objects in the similarity inheritance model have the property of *self sufficiency* from the programmer's perspective, meaning that every supported operation for an object and every piece of data it contains can be determined by examining the object's own definition rather than also requiring the inspection of parent objects or descriptive classes. The implication of the `L` element of the model is that the programmer, rather than working with abstract descriptions (as in the class-based model), creates and manipulates concrete descriptions, in which live objects reside. A significant difference in the concreteness just described from that of the prototype model is that the sharing relationships are created statically rather than dynamically.

From this model, the granularity difference between similarity inheritance and other approaches becomes clear. Both the class-based and prototype-based models support $\not\rightarrow$ at the granularity of methods (overriding), but neither supports $\rightarrow$ at the granularity of methods, e.g. the ability to inherit just one method. Multiple and mutual inheritance are direct by-products of fine-grained similarity. Multiple inheritance occurs in cases such as $A \rightarrow B$, and $C_{b1} \rightarrow B_{b1}$. Mutual inheritance occurs in cases such as $B_{b2} \rightarrow C_{b2}$ and $C_{b3} \rightarrow B_{b3}$.

5. Similarity inheritance in Forms/3

5.1 Interaction model

The interaction model is instantiated in the research spreadsheet VPL Forms/3 as follows. Operation `C` is supported by a *copy form* button, which copies the form selected in a scrolling list, and by a *paste* button on each form, which pastes selected cells onto the form. Recall that **C** is not simply an edit operation, but defines the → relationship. Operation **F** is supported by allowing the programmer to edit any formula that is visible. Liveness level **L** is level 4, so after every formula edit, immediate visual feedback is given about the edit's effect on the program. In Forms/3's representation **R**, each behavior (cell and formula), whether the result of sharing or not, is visible, which allows it to be edited by operation **F**. Shading indicates whether a form or cell is the result of sharing. Section 6 explains additional features of Forms/3's representation.

5.2 Large-grained inheritance

The Stack in Section 3 is an example of an object created from scratch. Multiple Stack objects can be created with the copy operation **C**, as was illustrated by the circle example. But suppose the programmer wants not another stack but something else that is similar to a stack, for example, a queue. Taking advantage of that similarity, the programmer can start with a copy of Stack, then modify the behavior using operation **F**. Inheritance and overriding of cell names as well as of cell formulas are allowed. A change to the *push* operation and some renaming of cells are all that is required to turn the copy into a Queue (see Figure 3).

Note that using similarity inheritance, the programmer identifies that two objects are simply similar in implementation or purpose. In contrast to this, in a class-based language, the programmer may spend extra time wondering what the "right" relationship is between a stack and a queue. One is not a subtype of the other, yet they are similar. In fact, extra work to reorder the inheritance hierarchy may be needed in some cases just to add one new class. As we saw above, however, similarity inheritance allows the programmer to create a similarity relationship without implying "is-a" subtype or subclass relationships. Instead, it defines a "like-a" relationship. For example, a queue is *like a* stack except that new items are inserted at the opposite end.

Because of the → relationship between Stack and Queue, changes to formula definitions on Stack will propagate to Queue unless they have been overridden. For example, a fix to the *push* operation on Stack would not have any effect on Queue, but a fix to the *pop* operation would propagate to the *dequeue* on the Queue form. Some prototype-based languages lose this ability to propagate changes to groups of objects because of their emphasis on object individuality; instead, shared parts must be abstracted out of the objects, as in Self's traits [20].

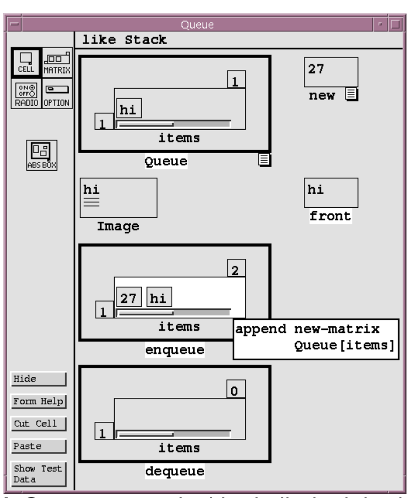

Figure 3: A Queue created with similarity inheritance from a Stack. Several names and one matrix are unshaded to indicate that they have been overridden. The *Image* cell could also be overridden to create a custom appearance for a Queue.

5.3 Fine-grained and multiple inheritance

As noted in the discussion of the model, the combination of large-grained and fine-grained similarity allows multiple inheritance. For example, in Forms/3 suppose a new form Deque is created via large-grained similarity from Queue. (A deque is a double-ended queue.) This new object needs to allow items to be added to either end of the queue. The programmer may notice that Stack's *push* is exactly the required behavior for Deque and can use fine-grained similarity, copying *push*, to allow Deque to inherit just that one operation from Stack. Because of the interaction element **R**, the programmer now sees the new cells as part of the definition of Deque also (Figure 4).

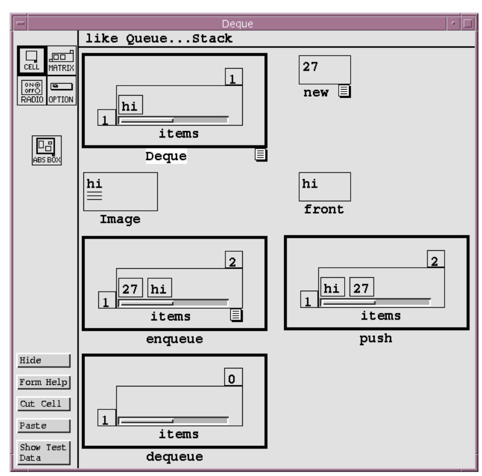

Figure 4: A Deque in progress, made by copying the Queue form and the *push* operation from Stack.

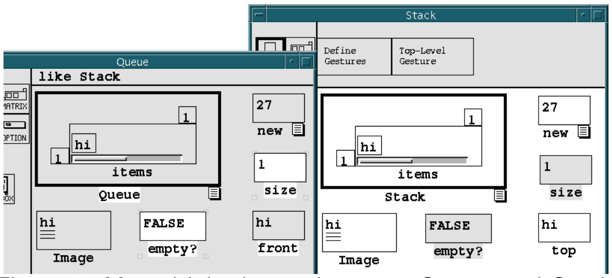

Figure 5: Mutual inheritance between Queue and Stack. The new cells *size* and *empty?* appear white on the Queue form where they originated, and shaded on the Stack form.

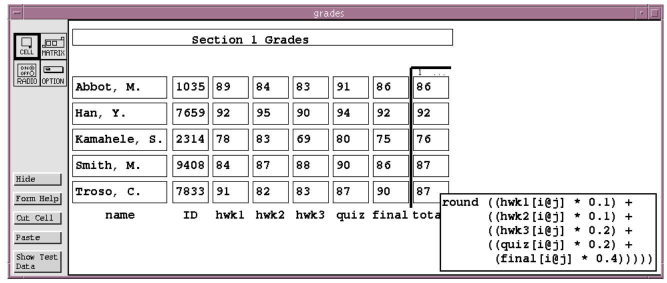

Figure 6: A spreadsheet to compute grades. The cells in the *total* column are grouped into a matrix and thus need only one formula (shown) to define their values. The formula computes the course grades via a weighted average.

Multiple inheritance in other languages can lead to conflicts when more than one method of the same name are unintentionally inherited. By providing inheritance on the level of cells, the similarity model allows the programmer to select only the operations that are actually needed, avoiding unintentional inheritance. (If the programmer does accidentally attempt to introduce a conflict, the system provides options for resolving it at the time of the edit.)

5.4 Mutual inheritance

Suppose, as in Figure 5, someone added the new operations *size* and *empty?* to Queue. Another programmer might find those operations useful for Stack as well and copy them to the Stack form. Stack and Queue now both inherit from each other. Like multiple inheritance, mutual inheritance is not a new concept in the language, but rather a feature of the flexibility of similarity inheritance, which makes mutual inheritance straightforward. To the best of our knowledge, similarity inheritance is the first model to support mutual inheritance.

5.5 An end-user example

In the previous sections, we have discussed our approach from the standpoint of how it can be used to share behavior among objects. However, as has been noted earlier, the approach is general enough to allow sharing of other pieces of programs, even when there is no relationship among the types of objects involved. This allows the same approach to be used for simple formula reuse as for object inheritance, instead of prior approaches, which relied on copy/paste and “replicate” options. The advantage of using inheritance for reusing spreadsheet formulas is that the relationships among originals and copies are *maintained*, supporting automatic propagation of bug fixes and explicit depiction of relationships.

For example, consider Figure 6, which shows a spreadsheet (written in Forms/3) to compute course grades. Suppose the user teaches several sections of the course, and keeps each section in a separate spreadsheet for convenience.

There are two reuse situations in this example: the reuse of the formula for the top row down through the remaining rows of this section, and the reuse of these formulas in other sections. In the first case, traditional spreadsheets use a “replicate” mechanism (copy down the rows). Our system does not apply inheritance to this case; instead, like some other spreadsheets, it has a way to group cells with a common formula. It is the second case in which we apply similarity inheritance. In the second case, traditional spreadsheets use a “copy/paste” mechanism (copying into other sections), and then if the weights need to be changed, the user would have to remember to do all of the copy/pasting again. However, if a system implements copying using similarity inheritance to make the relationships explicit, as does Forms/3, then a change to the weights in the first section can automatically propagate to all the other students.

As this example demonstrates, similarity inheritance can be used not only to maintain relationships among objects, types, and operations, but also among pieces of any sort of calculation. An attractive feature of this generality is that it affords a gradual migration path for users to move from using only simple numbers and strings in their formulas to using more complex objects with inheritance as they gain expertise, since the same mechanism for inheritance is employed for reusing formulas in both situations.

6. Explicit representation

The interaction model requires the existence of an explicit representation, **R**. To design the representation for similarity inheritance in Forms/3, we used a set of design benchmarks [23] that are a concrete application of several of the *cognitive dimensions* for programming systems [7]. To focus on the reasons behind our design choices for **R**, we present the representation from the perspective of the benchmarks that had the greatest influence on the design.

The *visibility of dependencies* benchmark is the ratio of program dependencies that are explicitly visible to the programmer. Green and Petre noted hidden dependencies as a severe source of difficulty in understanding programs. The dependencies of interest in the similarity inheritance model are those created by the → relationship. The dependencies can be at either the form or cell level. For each, the programmer may be interested in both “what affects this?” and “what does this affect?” for a total of four kinds of dependencies. Because of the importance of explicit representation to the similarity inheritance model, it was important that all four kinds of dependencies be explicitly represented.

The two “what affects this” questions are handled by legends. A legend under each formula lists the cell that it was directly copied from. If that cell was in turn copied from another, ellipsis follow and the name of the original cell is also given. The same legend mechanism is used for explicit representation at the form level (see Figure 7).

The two “what does this affect” questions are answered by optional copy dependency arrows (Figure 7) and by a summary view (Figure 8). The summary view represents each form as a node labeled with its name. Arrows between nodes make different kinds of relationships explicit. Our current design has three kinds of arrows indicating form copies, dataflow, and cell copies. The summary view is part of a package of “live” views we previously implemented that address other, non-inheritance oriented, reuse issues [21].

Visibility of program logic is another benchmark that influenced our design. The program logic of interest here is the logic of inherited formula. The current design

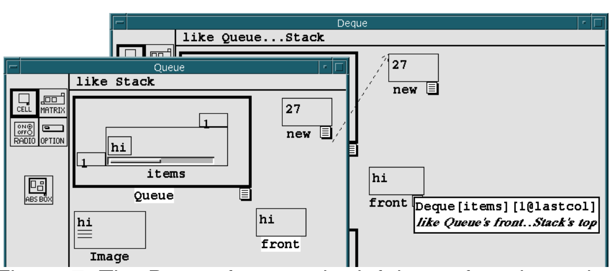

Figure 7: The Deque form on the left has a form legend at the top indicating it is copied from Queue which in turn was copied from Stack. (If there were intermediate forms, the legend would take the form “Queue...3...Stack” and the programmer could click on the 3 for a full list.) Deque’s *front* cell illustrates a formula legend. Copy dependencies among cells are also explicitly depicted with optional arrows such as the one from Queue’s *new* cell. (The formula legend in italics is not yet implemented and has been manually added to the screenshot.)

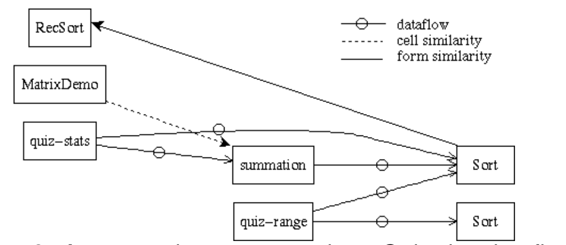

Figure 8: An example summary view. Only the dataflow arrows are implemented so far; the other two have been manually added to the screen shot.

makes an inherited formula visible in the place where it is inherited. This avoids the “yo-yo problem” in class-based languages where to see the program logic for a subclass, the programmer may need to visit several classes up and down the class hierarchy. The yo-yo problem is also exhibited in most prototype-based languages, where instead of class definitions, the programmer must examine multiple levels of parent objects to view inherited code.

Whenever features are added to a VPL’s representation, the limited size of the screen must be taken into account. Our representation is designed to fit into a small portion of the programmer’s screen. The display of form-level inheritance takes up just one line on the form, no matter how long the list becomes, because intermediate forms are normally elided. Likewise, cell-level inheritance legends take only one line per formula. These legends can also be hidden to conserve screen real estate when needed.

7. Discussion and future work

The similarity inheritance model was devised as a way to bring inheritance particularly to spreadsheet VPLs, but it may be suitable for other kinds of interactive VPLs as well, provided that they support the interaction model. We do not expect the approach to be used in strictly textual languages, which are usually defined apart from any environment, because this characteristic would seem to prevent any guarantee that the required elements of the interaction model would be present.

An important next step will be empirical work to learn whether users make fewer reuse errors or reuse formulas more often under similarity inheritance. Two other questions that we would like to explore are whether users are as comfortable with similarity inheritance as with copy/paste/replicate, and whether the explicit representation succeeds at making the flexibility inherent in the approach manageable. Finally, we would like to gather empirical data about whether and how people use mutual inheritance.

8. Conclusion

In this paper, we have presented a new model of inheritance for spreadsheet VPLs. The model supports large-grained inheritance, fine-grained inheritance, multiple inheritance, and mutual inheritance. The prototype implementation shows that the model can be incorporated into the spreadsheet paradigm, using only cells and declarative formulas, without violating the value rule or requiring users to learn other programming languages or macro languages.

We have shown also that the approach to inheritance can be used to manage reuse relationships among cells even in simple formula reuse, which traditionally has been supported only by copying or replicating a formula to other cells. This flexibility not only increases the support for this kind of operation, it also affords a gradual path for a user to progress from simple formula copy/paste to more advanced applications of the technique such as inheritance among user-defined types.

An important feature of similarity inheritance is object self-sufficiency. Self-sufficiency enables a concrete style of programming, and avoids the yo-yo problem. These features are a result of the semantic model's reliance on an interaction model. Although two of the four elements in the interaction model—the copy operation and the formula definition operation—could be accomplished textually, the other two elements—an explicit representation and a liveness level of 3 or higher—*require* visual elements in the environment that textual languages usually do not guarantee. Hence, it is unlikely that similarity inheritance would be possible for a textual language.

Acknowledgments

This work was supported in part by Hewlett-Packard, by Harlequin, by the National Science Foundation under grant CCR-9308649 and an NSF Young Investigator Award, and by a NASA Graduate Student Researcher Award.

References

[1] Atwood, J., M. Burnett, R. Walpole, E. Wilcox, S. Yang, "Steering programs via time travel," *IEEE Symp. on Visual Languages*, 4-11, Sept. 1996.

[2] Bracha, G. and W. Cook, "Mixin-based inheritance," *OOPSLA '90*, 303-311, Oct. 1990.

[3] Lee Braine and Chris Clack. "Object-flow," *IEEE Symp. on Visual Languages*, 418-419, Sept. 1997.

[4] Burnett, M. and H. Gottfried, "Graphical definitions: Expanding spreadsheet languages through direct manipulation and gestures," *ACM Trans. on Computer-Human Interaction*, March 1998.

[5] Burnett, M., A. Goldberg, T. Lewis, (eds.), *Visual Object-Oriented Programming Concepts and Environments*. Prentice-Hall/Manning Pubs., 1995.

[6] Cox, P., F. Giles, T. Pietrzykowski. "Prograph: a step towards liberating programming from textual conditioning," *IEEE Workshop on Visual Languages*, 150-156, Oct. 1989.

[7] Green, T. and M. Petre, "Usability analysis of visual programming environments: A 'cognitive dimensions' framework," *J. Visual Languages and Computing*, 131-174, June 1996.

[8] Hudson, S., "User interface specification using an enhanced spreadsheet model," *ACM Trans. on Graphics*, 209-239, July 1994.

[9] Kay, A., "Computer software," *Scientific American,* 53-59, Sept. 1984.

[10] Lewis, C., "NoPumpG: Creating interactive graphics with spreadsheet machinery," in *Visual Programming Environments: Paradigms and Systems* (E. Glinert, ed.), IEEE CS Press, 526-546, 1990.

[11] Lieberman, H., "Using prototypical objects to implement shared behavior in object oriented systems," *OOPSLA*, Nov. 1986, 214-223.

[12] Myers, B., "Graphical techniques in a spreadsheet for specifying user interfaces," *CHI '91,* 243-249, May 1991.

[13] Myers, B., et al., "Garnet: Comprehensive support for graphical, highly interactive user interfaces," *Computer,* 71-85, Nov. 1990.

[14] Ng, K. and C. Luk. "I+: A multiparadigm language for object-oriented declarative programming," *Computer Languages* 21(2), 81-100, 1995.

[15] Penz, F., "Visual programming in the ObjectWorld," *J. Visual Languages and Computing* 2(1), 17-41, March 1991.

[16] Piersol, K., "Object oriented spreadsheets: The Analytic Spreadsheet Package," *OOPSLA '86*, 385-390, Sept. 1986.

[17] Smedley, T., P. Cox, S. Byrne, "Expanding the utility of spreadsheets through the integration of visual programming and user interface objects," *Adv. Visual Interfaces '96*, 148-155, May 1996.

[18] Taivalsaari, A., "A critical view of inheritance and reusability in object-oriented programming," Ph.D. Thesis, Univ. of Jyvaskyla, 1993.

[19] Tanimoto, S., "VIVA: A visual language for image processing," *J. Visual Languages and Computing* 2(2), 127-139, June 1990.

[20] Ungar, D., C. Chambers, B. Chang, U. Hölze, "Organizing programs without classes," *J. Lisp and Symbolic Computation* 4(3), 1991.

[21] Walpole, R. and M. Burnett, "Supporting reuse of evolving visual code," *1997 IEEE Symposium on Visual Languages*, 68-75, Sept. 1997.

[22] Wilde, N. and C. Lewis, "Spreadsheet-based interactive graphics: From prototype to tool," *CHI '90 Conference on Human Factors in Computing Systems*, 153-159, Apr. 1990.

[23] Yang, S., M. Burnett, E. DeKoven, M. Zloof, "Representation design benchmarks: A design-time aid for VPL navigable static representations," *J. Visual Languages and Computing* 8(5/6), Oct./Dec. 1997.