Oregon State University, TR 97-60-15, November 1997.

Similarity Inheritance: A New Model of Inheritance for Spreadsheet Languages

Rebecca A. Walpole and Margaret M. Burnett
Department of Computer Science
Oregon State University, Corvallis, Oregon 97331
{walpolr, burnett}@cs.orst.edu (541) 737-2539

ABSTRACT

Although spreadsheets can be argued to be the most widely-used end-user programming languages today, they are very limited compared to other programming languages, 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 a new approach to inheritance, similarity 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 for a research spreadsheet language. We show that bringing inheritance functionality to the spreadsheet paradigm supersedes previous support for reuse in spreadsheets, and does so without sacrificing the concreteness, flexibility, and directness that characterize spreadsheets.*

KEYWORDS: inheritance, spreadsheet languages, visual programming languages, end-user programming

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 they change a reused formula. 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 languages.

We use the term *spreadsheet languages* to refer to a variety of systems that follow the spreadsheet paradigm, from commercial spreadsheets to more sophisticated research systems that follow the declarative, one-way constraint evaluation model. The essence of the paradigm is summarized by Alan Kay's *value rule* for spreadsheets [Kay 1984], 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 languages, and an instantiation of the approach in the research spreadsheet language Forms/3 [Burnett and Ambler 1994; Atwood et al. 1996; Gottfried and Burnett 1997]. The approach, called *similarity inheritance*, provides a concrete way of sharing behavior among objects in a spreadsheet language. The unique attributes of similarity inheritance are that:

- it provides a complete, explicit, 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 languages 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 Combining Spreadsheets with Object-Oriented Programming

Spreadsheets and more advanced spreadsheet languages have had little work to date on approaches to inheritance, perhaps because there has been only a little work that incorporates support for objects. 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 incorporation of 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 languages have also incorporated external languages to support object-oriented features. ASP (Analytic Spreadsheet Package) is a spreadsheet language in which every cell can be any object, and every formula is written in Smalltalk code [Piersol 1986]. Smedley, Cox, and Byrne have incorporated the visual programming language Prograph and user interface objects into a conventional spreadsheet for GUI programming [Smedley et al. 1996]. Both of 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.

C32 [Myers 1991] is a spreadsheet language that is part of the Garnet and Amulet user interface development environments [Myers et al. 1990; Myers et al. 1996]. C32 uses graphical techniques along with inference to specify constraints in user interfaces. C32 does not itself feature the graphical creation and manipulation of objects. Instead, this function is performed by another part of the Garnet/Amulet package. The combination of tools in the Garnet/Amulet package features strong support for programming with built-in GUI objects via visual techniques, but does not support any other kinds of objects, which must be written and manipulated in Lisp/C++.

Some research spreadsheet languages have moved toward expanding the types of objects supported without the use of external programming languages. One of the pioneering systems in this direction was NoPumpG [Lewis 1990] and its successor NoPumpII [Wilde and Lewis 1990], two spreadsheet languages 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 complex or user-defined objects.

Penguims [Hudson 1994] is a spreadsheet language 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.

2.2 Prototype-Based Inheritance

Traditionally, object-oriented inheritance is a sharing mechanism between a class and its subclass. The most prevalent alternative to this class-based inheritance has been prototype-based inheritance, and our approach is most like this alternative.

Inheritance in prototype-based languages is based on concrete parent objects rather than abstract classes. In most prototype-based languages, inheritance is accomplished through *delegation*. With delegation, 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 [Ungar et al. 1991] is perhaps the best-known prototype-based language.

ObjectWorld [Penz 1991; Penz and Wollinger 1993] is a prototype-based language that, like ours, uses visual mechanisms to emphasize concreteness and does not use delegation. However, unlike our approach, ObjectWorld does not use any inheritance mechanism, instead achieving code reuse through object composition combined with automatic message propagation. An important difference between our similarity inheritance approach and most prototype-based languages (including Self and ObjectWorld) is that our model does not use any sort of message passing.

Kevo [Taivalsaari 1993] is one of the few prototype-based languages that does not use message passing. Kevo 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 there are no change propagation mechanisms for objects outside the clone family. Kevo approximates multiple inheritance and fine-grained inheritance via a cut/copy/paste metaphor, but changes to the original code do not propagate, and must be recopied and pasted by the programmer.

2.3 Fine-Grained Inheritance

Although most approaches to inheritance operate at the granularity of entire classes or objects, there is some research exploring inheritance at finer granularities. Mixins [Bracha and Cook 1990] are a technique for providing inheritance on a more fine-grained scale than whole classes. Also known as abstract subclasses, mixins are partial classes that 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^+ [Ng and Luk 1995]. I^+ inheritance is not determined by subclassing, but by explicitly listing the methods to inherit.

The fine-grained aspects of similarity inheritance are closer to the I^+ approach than to the mixin approach. Some differences however, are that similarity inheritance allows even finer-grained inheritance than methods, and is flexible enough to allow mutual inheritance. Also, our approach is particularly focused on maintaining attributes important to spreadsheet languages, such as concreteness and immediate visual feedback, attributes that are not present in I^+.

3. BACKGROUND: INTRODUCTION TO FORMS/3

We have created a prototype implementation of our approach to inheritance in the spreadsheet language Forms/3, and the examples in this paper are presented in that language. This section provides the necessary background in Forms/3 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 (see Table 1).

Forms/3 has long supported an extensible collection of

formula	::=	*BLANK* | expr
expr	::=	*CONSTANT* | ref | infixExpr | prefixExpr | ifExpr
infixExpr	::=	subExpr infixOperator subExpr
prefixExpr	::=	unaryPrefixOperator subExpr | binaryPrefixOperator subExpr subExpr
ifExpr	::=	IF subExpr THEN subExpr ELSE subExpr | IF subExpr THEN subExpr
subExpr	::=	*CONSTANT* | ref | (expr)
infixOperator	::=	+ | - | * | / | AND | OR | = | ...
unaryPrefixOperator	::=	ROUND | CIRCLE | ...
binaryPrefixOperator	::=	APPEND | ...
ref	::=	*CELL* | *MATRIX* | *ABS* | *ABS* [*CELL*] | *MATRIX* [subscripts] | *ABS* [*MATRIX*] | *ABS* [*MATRIX*] [subscripts]
subscripts	::=	matrixSubscript@matrixSubscript
matrixSubscript	::=	expr

Table 1: The grammar for the formula language in this paper. There are also 6 "pseudo references"—I, J, NUMROWS, NUMCOLS, LASTROW, and LASTCOL—that are used in matrix subscripts. Including these in the grammar is straightforward but tedious, and we have omitted them for brevity.

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 [Gottfried and Burnett 1997], such as by sketching a new circle or by stretching an existing circle by direct manipulation. The graphical way is a shortcut for the first way, and we will use only the first way 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 which, as one would expect on a spreadsheet, responds with immediate visual feedback as each new formula is entered. The formulas specify the internal data of the object, how it should appear visually on the screen, and any operations that it provides.

The internal data is defined by cells and matrices that can be placed inside *abstraction boxes*, which 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 abstraction box *Stack*), in which the programmer has added 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 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.

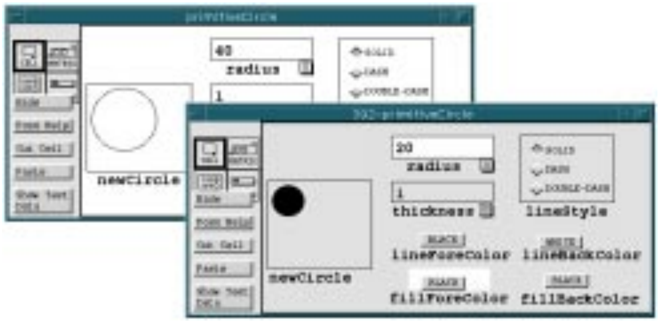
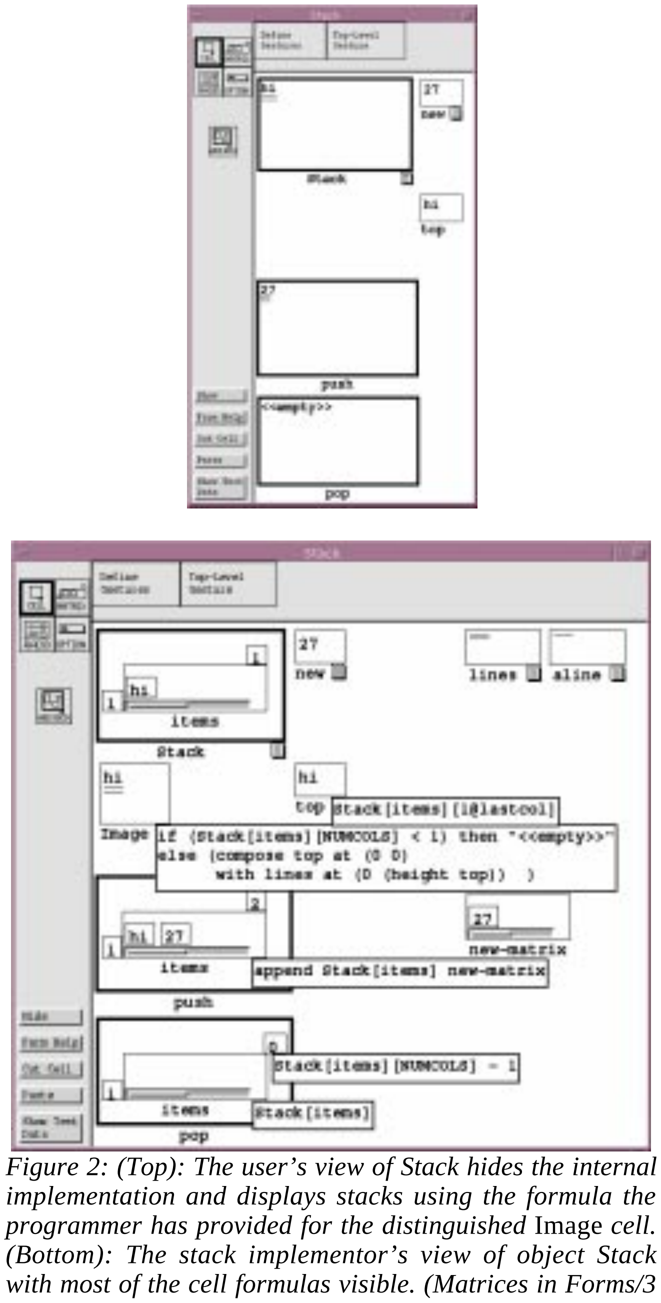


Figure 1: The white primitiveCircle *form is a built-in form that defines a prototypical instance of type primitiveCircle. The gray* 392-primitiveCircle *form is a copy that has been modified to describe a different instance. The circle in cell* newCircle *is defined by the other cells, which specify 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.*



```
Stack[items][1@lastcol]
```

```
if (Stack[items][NUMCOLS] < 1) then "<<empty>>"
else (compose top at (0 0)
      with lines at (0 (height top)) )
```

```
append Stack[items] new-matrix
```

```
Stack[items][NUMCOLS] - 1
```

```
Stack[items]
```

Figure 2: (Top): The user's view of Stack hides the internal implementation and displays stacks using the formula the programmer has provided for the 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, independent 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 defined by the tuple:

$$(\chi, \delta, \lambda, \rho)$$

where χ is the copy operation that creates a shared definition, δ is the formula definition operation, λ is a liveness level 3 or higher from Tanimoto's liveness scale [Tanimoto 1990] indicating that immediate semantic feedback is automatically provided[1], and ρ is a representation mechanism that explicitly includes all shared formulas and relationships in the representation of each object.

Two 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) and (2) it does not necessarily map to a static textual syntax (for example, it allows dynamic syntaxes). 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.

[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 semantic model can now be defined as follows. Each object O in a program is a set of definitions $\{O_{d1}, O_{d2}, \ldots, O_{dn}\}$. Each O_{di} is a formula residing in a cell. The symbol $\rightarrow$ (pronounced "shares with") indicates a shared definition; the arrow points from the original version to the copied one. The semantics of $\rightarrow$ are

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

The operations χ and δ determine when $\rightarrow$ holds, as summarized in the following table.

	Operation	Precondition	Postcondition
(1)	χ applied to A	A is an existing object; B does not exist	B is a new object and $\forall i$, $A_{di} \rightarrow B_{di}$
(2)	χ applied to A_{di} and B	A and B are existing objects, $A \neq B$	$A_{di} \rightarrow B_{di}$
(3)	δ applied to B_{di}	B is an existing object	$\forall A$, $A_{di} \not\rightarrow B_{di}$

Row (1) defines large-grained similarity, and means that if χ is applied to an object A, a similarity relationship will be created between A and a new object B such that all of A's definitions share with B (this can be abbreviated $A \rightarrow B$). Row (2) defines fine-grained similarity, which allows a single definition A_{di} to be copied to object B to create a shared relationship between A_{di} and B_{di}. Row (3) implies that overriding removes any "upstream" sharing relationships, but not "downstream" relationships

Due to element ρ 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 itself rather than also requiring the inspection of parent objects or descriptive classes. The implication of the λ element of the model is that the programmer creates and manipulates live objects while constructing the program, rather than abstract descriptions of objects.

From this model, differences between similarity inheritance and other approaches become clear. For example, the class-based model has a $\rightarrow$ relationship defined between classes (not objects) and does not have an interaction model. Prototype-based inheritance, on the other hand, may contain part or all of the interaction model, but does not define the

$\rightarrow$ relationship between copied objects. Similarity inheritance is also different from both models in that it allows not only multiple inheritance but also mutual inheritance. Multiple inheritance occurs in cases such as $A\rightarrow B$, $C_{d1}\rightarrow B_{d1}$, $C_{d2}\rightarrow B_{d2}$ and $C_{d3}\rightarrow B_{d3}$. Mutual inheritance occurs in cases such as $B_{d2}\rightarrow C_{d2}$ and $C_{d3}\rightarrow B_{d3}$.

5. SIMILARITY INHERITANCE IN FORMS/3

5.1 Interaction Model

The interaction model is instantiated in the research spreadsheet language Forms/3 as follows. An object in Forms/3 is a form. Operation χ 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. Operation δ is supported by allowing the programmer to edit any formula that is visible. Liveness level λ 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 ρ, each definition (cell), whether copied or not, is visible, which allows it to be edited by operation δ. Shading indicates whether a form or cell is copied. Section 6 explains additional features of Forms/3's representation.

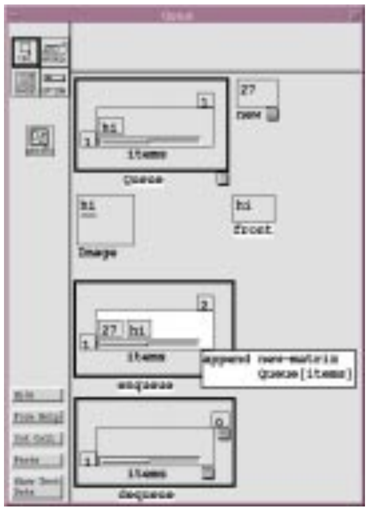


Figure 3: A Queue created with similarity inheritance from a Stack. Several names and one cell are unshaded to indicate that they have been overridden. The image could also be overridden to create a custom appearance for a Queue.

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 χ, 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 δ. A change to the *push* operation and some renaming of cells is all that is required to turn the copy into a Queue (see Figure 3). In this prototype, inheritance and overriding of cell names as well as of cell formulas are allowed.

Note that using similarity inheritance, the programmer simply identifies that two objects are 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, and 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 $\rightarrow$ 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 affect on Queue, but a fix to the *pop* operation would propagate to the *dequeue* on the Queue form. Most prototype-based languages lose this ability to propagate changes to groups of objects because of their emphasis on object individuality; instead, shared parts are abstracted out of the objects.

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 ρ, the programmer now sees the new cells as part of the definition of Deque also (Figure 4).

Multiple inheritance in other languages can lead to conflicts when more than one method of the same name are

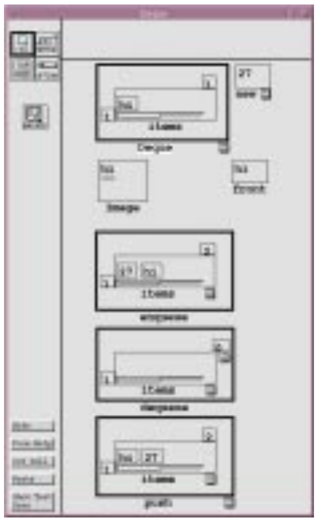


Figure 4: A Deque in progress made by copying the Queue form and the push operation from Stack.

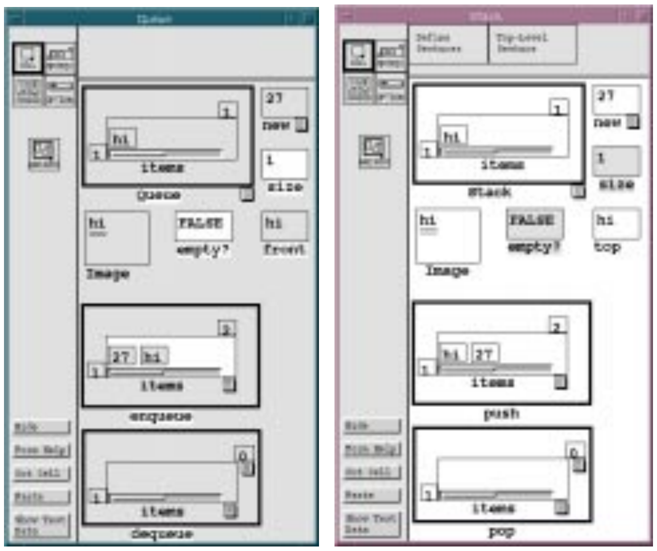


Figure 5: Mutual inheritance between Stack and Queue. The new cells size *and* empty? *appear white on the Queue form where they originated and shaded on the Stack form.*

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 a feature of the flexibility of similarity inheritance, not a new concept in the language, which makes mutual inheritance straightforward. To the best of our knowledge, similarity inheritance is the first model to support mutual inheritance.

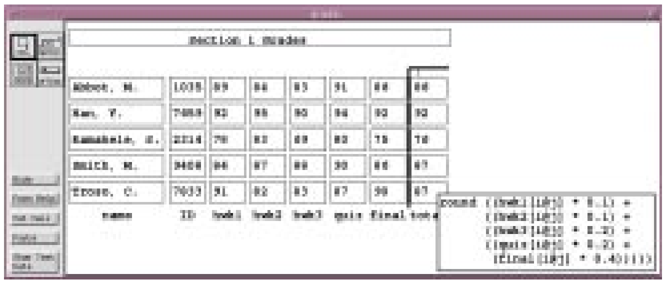
5.5 An End-User Example

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 to using inheritance for reusing spreadsheet code 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 just to maintain relationships among objects, types, and operations, but also among pieces of any sort of code. 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 code in both situations.



Section 1 Grades

name	ID	hwk1	hwk2	hwk3	quiz	final	total
Abbot, M.	1035	89	84	83	91	88	88
Kan, Y.	7058	92	95	90	94	92	92
Kamakele, E.	2314	78	82	69	82	75	78
Smith, M.	3408	84	87	89	90	86	87
Trose, C.	7033	91	82	83	87	90	87

```
round ((hwk1[1@]] * 0.1) +
       ((hwk2[1@]] * 0.1) +
       ((hwk3[1@]] * 0.2) +
       ((quiz[1@]] * 0.2) +
        (final[1@]] * 0.4))))
```

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.*

6. EXPLICIT REPRESENTATION

The fourth element of the interaction model, ρ, requires the existence of an explicit representation exists, but does not specify the form it should take. In the process of designing the representation for Forms/3, we used a set of design benchmarks [Yang et al. 1997] that are a concrete application of several of the *cognitive dimensions* for programming systems by researchers from the field of cognitive psychology [Green and Petre 1996]. The cognitive dimensions provide a foundation for considering the cognitive issues of representing programs, and provide an increment in formality over previous ad-hoc methods. We were able to use the benchmarks to iteratively improve earlier drafts of this design. For brevity, we will discuss only the benchmarks that had the greatest influence on the final 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 (see Figure 7). The same legend mechanism is used for explicit representation at the form level (also in Figure 7).

The two “what does this affect” questions are answered by 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. Only dataflow arrows are implemented so far. The summary view is part of a package of “live” views we previously implemented that address other, non-inheritance oriented, reuse issues [Walpole and Burnett 1997].

Visibility of program logic is another benchmark that influenced our design. The program logic of interest here is the logic of the inherited code. The current design makes inherited code visible in the place where it is inherited. This is a significant improvement over the “yo-yo problem” encountered 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 object definitions to view inherited code.

Whenever features are added to a visual programming language’s representation, the limited amount of screen real estate available 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 elided. Likewise, for cell-level inheritance feedback takes only one line per formula. These legends can also be hidden to conserve screen real estate when needed.

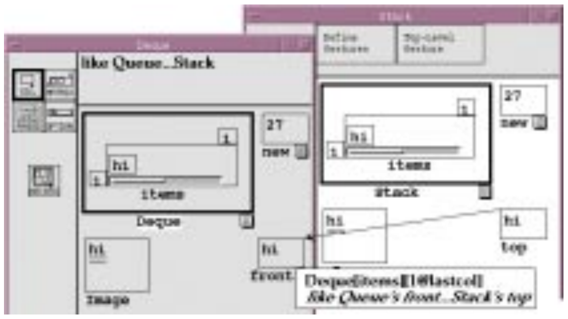


Figure 7: Superimposed legends and copy dependency arrow. The Deque form on the left has a form legend at the top indicating it is copied from Queue which in turn was copied from Stack. (If there were intermediate forms, the legend would take the form “Queue...3...Stack” and the programmer could click on the 3 for a full list.) Deque’s front *cell illustrates a formula legend. Copy dependencies among cells can also be explicitly depicted with arrows such as the one from Stack’s* top *cell.*

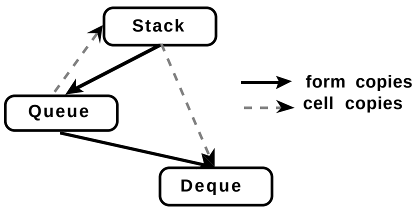


Figure 8: An example summary view displaying the relationships among the Stack, Queue and Deque forms.

7. DISCUSSION AND FUTURE WORK

The similarity inheritance model was devised as a way to bring inheritance particularly to spreadsheet languages, but it may be suitable for other 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 feature would seem to prevent any guarantee that the required elements of the interaction model would be present.

Our research prototype runs on Sun and Hewlett-Packard color workstations using Harlequin’s Liquid Common Lisp and the Garnet user interface development environment [Myers et al. 1990]. The prototype currently includes all the features described except some of the representation features described in the previous section and the user interface with which the programmer resolves name conflicts.

When the rest of the prototype is completed, we plan empirical work to learn whether users make fewer reuse errors or reuse code 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 languages. 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.

Since the flexibility of the approach would be unwieldy without strong support by the representation scheme, the approach includes a representation that explicitly depicts the reuse relationships, eliminating the need for the user to remember and manually track these relationships. Further, because the representation is explicit, the yo-yo problem is avoided.

Finally, we have shown that the approach to inheritance can be used to improve the way reuse relationships among cells are managed even in simple formula reuse, which has been traditionally supported only by copying or replicating a formula to other cells. This flexibility not only improves the support for this kind of operation, it also allows for a straightforward 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.

ACKNOWLEDGMENTS

We would like to thank the members of our research group for their work on the Forms/3 design and implementation and for their feedback on this research. 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

[Astudillo 1996] Astudillo, H., “Reorganizing Split Objects,” *OOPSLA ‘96*, 138-149, 1996.

[Atwood et al. 1996] Atwood, J., M. Burnett, R. Walpole, E. Wilcox, and S. Yang, “Steering Programs via Time Travel,” *1996 IEEE Symposium on Visual Languages*, 4-11, 1996.

[Bracha and Cook 1990] Bracha, G. and W. Cook, “Mixin-Based Inheritance,” *OOPSLA ‘90*, 303-311, 1990.

[Burnett and Ambler 1994] Burnett, M. and A. Ambler, “Interactive Visual Data Abstraction in a Declarative Visual Programming Language,” *Journal of Visual Languages and Computing*, 29-60, 1994.

[Gottfried and Burnett 1997] Gottfried, H. and M. Burnett, “Graphical Definitions: Making Spreadsheets Visual through Direct Manipulations and Gestures,” *1997 IEEE Symposium on Visual Languages*, 246-253, 1997.

[Green and Petre 1996] Green, T. and M. Petre, “Usability Analysis of Visual Programming Environments: A ‘Cognitive Dimensions’ Framework," *Journal of Visual Languages and Computing*, 131-174, 1996.

[Hudson 1994] Hudson, S., “User Interface Specification Using an Enhanced Spreadsheet Model,” *ACM Trans. on Graphics*, 209-239, 1994.

[Kay 1984] Kay, A., “Computer Software,” *Scientific American*, 53-59, 1984.

[Lewis 1990] 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.

[Myers 1991] Myers, B., “Graphical Techniques in a Spreadsheet for Specifying User Interfaces,” *CHI ‘91*, 243-249, 1991.

[Myers et al. 1990] Myers, B., et al., “Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces,” *Computer*, 71-85, Nov. 1990.

[Myers et al. 1996] Myers, B., R. Miller, R. McDaniel, and A. Ferrency. “Easily Adding Animations to Interfaces Using Constraints,” *ACM Symposium on User Interface Software and Technology*, 119-128, 1996.

[Ng and Luk 1995] Ng, K. W. and C. K. Luk. “I+: A Multiparadigm Language for Object-Oriented Declarative Programming,” *Computer Languages* 21(2), 81-100, 1995.

[Penz 1991] Penz, F., “Visual Programming in the ObjectWorld,” *Journal of Visual Languages and Computing* 2(1), 17-41, 1991.

[Penz and Wollinger 1993] Penz, F. and T. Wollinger, “The ObjectWorld, a Classless, Object-Based, Visual Programming Language,” *OOPS Messenger* 4(1), 26-35, 1993.

[Piersol 1986] Piersol, K., “Object Oriented Spreadsheets: The Analytic Spreadsheet Package," *OOPSLA ‘86*, 385-390, 1986.

[Smedley et al. 1996] Smedley, T., P. Cox, and S. Byrne, “Expanding the Utility of Spreadsheets through the Integration of Visual Programming and User Interface

Objects," *Advanced Visual Interfaces '96*, 148-155, 1996.

[Taivalsaari 1993] Taivalsaari, A., "A Critical View Of Inheritance and Reusability in Object-Oriented Programming," Ph.D. Thesis, University of Jyvaskyla, 1993.

[Tanimoto 1990] Tanimoto, S., "VIVA: A Visual Language for Image Processing," *Journal of Visual Languages and Computing* 2(2), 127-139, 1990.

[Ungar et al. 1991] Ungar, D., C. Chambers, B. Chang, and U. Hölze, "Parents Are Shared Parts of Objects: Inheritance And Encapsulation In Self," *Journal of Lisp and Symbolic Computation* 4(3), 1991.

[Walpole and Burnett 1997] Walpole, R. and M. Burnett, "Supporting Reuse of Evolving Visual Code," *1997 IEEE Symposium on Visual Languages*, 68-75, 1997.

[Wilde and Lewis 1990] Wilde, N. and C. Lewis, "Spreadsheet-Based Interactive Graphics: From Prototype to Tool," *CHI '90 Conference on Human Factors in Computing Systems*, 153-159, 1990.

[Yang et al. 1997] Yang, S., M. Burnett, E. DeKoven, and M. Zloof, "Representation Design Benchmarks: A Design-Time Aid for VPL Navigable Static Representations," *Journal of Visual Languages and Computing*, 1997 (to appear).