Journal of Visual Languages and Computing 13(4), Ausust 2002, pp. 421-447.

End-User Programming of Time as an "Ordinary" Dimension in Grid-Oriented Visual Programming Languages*

Margaret Burnett, Nanyu Cao, Miguel Arredondo-Castro, and John Atwood
Department of Computer Science
Oregon State University, Corvallis, Oregon 97331
{burnett, caona, arredomi, atwoodj}@cs.orst.edu

Abstract

Specifying varying speeds and temporal relationships is necessary when programming graphical animations, but support for temporal programming has usually been done by adding new language features to a VPL, and these features must be mastered over and above the other aspects of the VPL. However, some researchers have believed that time should be able to be treated like just another dimension. In this paper, we explore whether programming of temporal relationships can be supported using exactly *the same devices as in spatial programming in grid-oriented VPLs. Toward this end, we provide a continuum of models aimed at this goal, with their advantages and disadvantages, and identify core issues that help illuminate the essence of the problem. We also describe an empirical study comparing one of these models against a widely-used traditional approach. The results were that end users were significantly more successful using the one model tested than with the traditional approach.*

1. Introduction

Many programming languages do not provide explicit control to the programmer over the temporal aspects of a program. However, there are important applications in which temporal behavior is critical to a program's correctness, especially in applications involving visual data. One example is animated graphics, a popular domain for visual programming languages (VPLs).

In earlier work in Forms/3 [Burnett and Gottfried 1998, Burnett et al. 2001] we showed how adding a time dimension to the spreadsheet paradigm facilitates programming of animated graphics [Atwood et al. 1996, Burnett et al. 1998, Carlson et al. 1996]. Our interest was in the realm of software animation, and we showed a straightforward way to program an animation such as the one in Figure 1. However, another goal of algorithm animation is to program such an animation without modifying the original (sorting) algorithm. Although we had initially believed taking this step would

*An early version of this paper previously appeared in conference form [Burnett et al. 2000].

be straightforward, we later discovered this not to be the case. Figure 2 shows the sorting algorithm as programmed using the stream-based approach found in many traditional languages. As Figure 2 demonstrates, two problems arose in programming the temporal relationships between the sort and its animation:

Problem 1: How to program temporal behaviors in a notation closer to how values evolve over time than in the traditional, one-dimensional syntax of `earlier` and `fby`.

Problem 2: How to program temporal interrelationships without artificially inserting code such as `counter`, whose purpose is to slow down the algorithm.

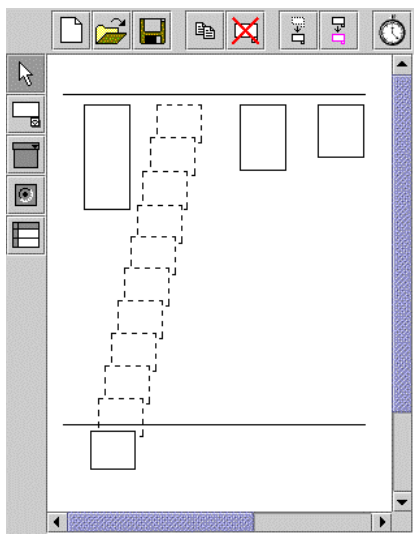


Figure 1: The animation of a sorting algorithm in Forms/3. A moving box drops down gradually (10 steps) into place when its corresponding element is inserted into a sorted group. This screenshot has been annotated with the dotted entries to show movement over time.

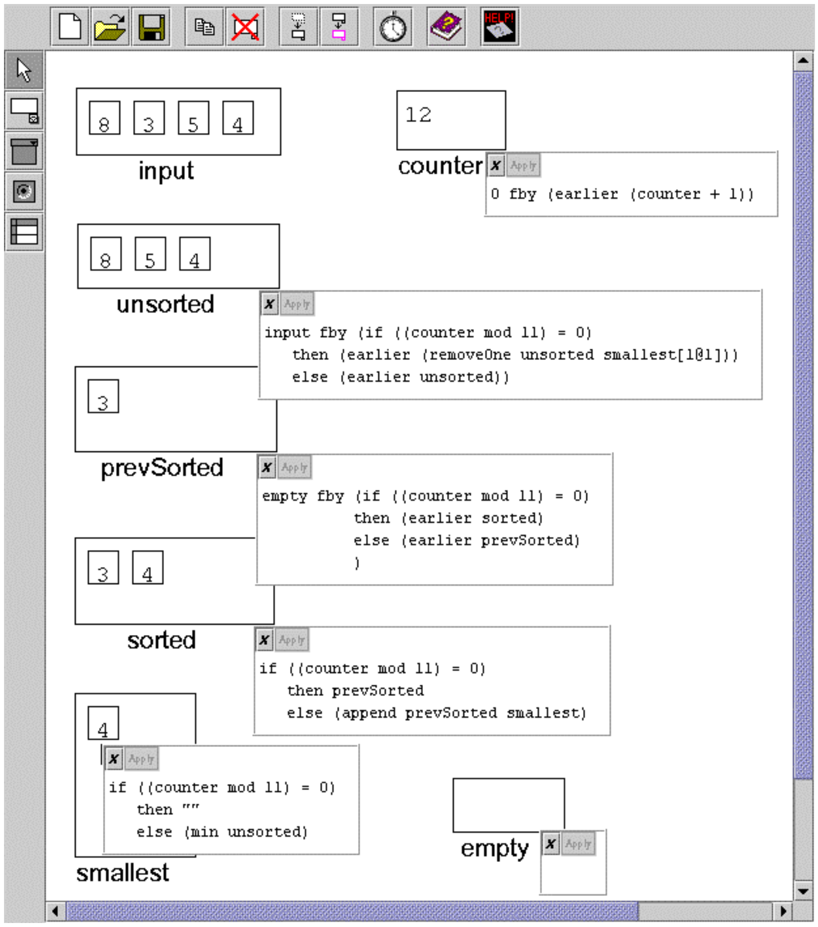


Figure 2: This selection sort, written using a previous version of Forms/3, required cell `counter` to slow down the algorithm so that the accompanying animation of Figure 1 had time to go through 10 graphical movements after each element's insertion. The formulas on the `input` grid are simply constant values; all other formulas are explicitly shown. (Forms/3 formulas can be displayed simultaneously with values, as shown; to close up a formula subwindow, a user can click on the "x" icon.)

In this paper, we explore whether visually programming temporal behavior can be done via exactly the same techniques used in visually programming spatially. Although visualization researchers have explored new devices for controlling visualization speed, our interest is to *avoid* new devices while enhancing control over *program* speed. We will focus on ways to programmatically control speed and temporal interrelationships among elements of the program, such as are needed in programming animated graphics, but will ignore the issue of programming temporal deadlines. (The latter is important in some domains, but is not required for most animated graphics programming.) Our interest is generally in understandability of temporal programming, but we are especially interested in supporting such programming for end users (users with no programming training).

We will consider this issue in grid-oriented VPLs. These include members of the spreadsheet paradigm, as well as other grid-oriented VPLs in which programming makes significant use of the positions of values or objects in a spatial grid. The issue is particularly relevant to VPLs of this group supporting grid-based programming of graphics, such as AgentSheets [Ioannidou and Repenning 1999], Cocoa (now known as Stagecast) [Heger et al. 1998], Forms/3, NoPump [Wilde and Lewis 1990], Spreadsheet for Information Visualization [Chi et al. 1998], and Spreadsheet for Images [Levoy 1994].

The contributions of this paper are three. First, several models that address variants of this issue are presented. Second, differences between temporal and spatial programming are identified that can cause difficulties in supporting grid-based programming of temporal behavior, because they can lead to language design problems that are known to interfere with humans' programming abilities. Third, we present empirical data comparing one of the visual grid-based models with the traditional one-dimensional stream-based approach. The results were that the one model which was tested was significantly more effectively used by end-user programmers than the traditional approach.

2. Background and Related Work

2.1 Cognitive Dimensions

It is possible to bring research into cognitive issues of programming to bear upon VPL design decisions by considering Green's and Petre's *Cognitive Dimensions*, a distillation of psychology of programming knowledge into a form usable by non-psychologists [Green and Petre 1996]. This work has motivated some of our design goals. For example, solving Problem 1 would improve *closeness of mapping*, the similarity of the programming notation to the problem being solved.

Consistency is another of the Cognitive Dimensions particularly pertinent to this work: "When some of the language has been learnt, how much of the rest can be inferred?" [Green and Petre 1996]. Since (spatial) grid-based programming in spreadsheets has been shown to be easy enough for millions of end users to use, a level of consistency allowing those users to reapply the same techniques to time suggests that users would be able to master temporal programming as well, and this is what motivated our interest in this strategy.

In this paper, we will use the Cognitive Dimensions as an early evaluation mechanism for design alternatives. Cognitive Dimensions are useful for raising warnings when a design decision is potentially at odds with research from cognitive principles of programming. However, it should be noted that Cognitive Dimensions cannot "prove" that no such problems remain. Thus, Cognitive Dimensions are best used as complements to other evaluation devices such as empirical studies.

2.2 VPLs and Temporal Behavior

There are several non-VPL animation systems (e.g. Balsa [Brown 1988], Tango [Stasko 1989], and Dance [Stasko 1991], which is the visual front-end to Tango) that support programming *temporal behavior of visualizations*, but not of the program itself. Also, although there are several VPLs that support *visualization of temporal behavior* (e.g., [Duecker et al. 1997, Freeman et al. 1996]), these VPLs have not explicitly supported *programming* of the program's temporal behavior.

Our interest is in programming temporal behavior of the programs themselves. Some early VL work regarding programming temporal aspects of software was done by supporting temporal logic in program specifications. The earliest such VL approach is probably GIL (Graphical Interval Logic) [Kutty et al. 1993]. GIL is a visual temporal logic for reasoning about the temporal properties of programs. It is a high-level, proof-oriented approach aimed primarily at visual specification and subsequent comparison to a (separate) program's properties. While there is also a tool supporting generation of a model from GIL temporal specifications, it does not support programming other than of temporal specifications.

There have been numerous examples of VPL research about coding temporal information via devices specific to temporal programming. Duisberg's work with temporal constraints[1] [Duisberg 1986] and with the use of music notation devices [Duisberg 1990] are perhaps the earliest VPL work aimed at temporal characteristics of animated graphics. Later examples include programmatic use of diagrams oriented toward time, such as space-time diagrams and concurrency maps as in [Stankovic and Zhang 1997] and time lines as in [Hibino and Rundersteiner 1997].

Treating time as temporal events has been another approach. For example, in Pavlov [Wolber 1997], there is an integer global time. Programmers explicitly specify what will happen at time n or every n times. However, it is not possible to refer to other objects' previous states. Pavlov's temporal programming is unlike Pavlov's space programming because unlike the other aspects of Pavlov, temporal information is not inferred from demonstrations.

There has also been significant work on temporal aspects of programming multimedia applications. Like the work described above, this body of work also provides constructs specific to temporal programming. One of the earliest examples of such work in VPLs was TYRO [MacNeil 1991], a demonstrational, constraint-oriented multimedia authoring language. TYRO is intended as a "designer's apprentice," and infers the correct temporal layouts from a combination of constraints and examples programmed using temporal-specific mechanisms. More recently, Song et al. developed an elastic time model, in which the deadline constraints are specified as the endpoints of a "spring" and the system uses these as constraints to schedule the computations that need to fit in this spring [Song et al. 1996]. Their time-box representation provides a visual way of specifying these spring endpoints and their relationships to other springs.

[1] Temporal constraints have later been used in non-visual approaches to animations, including Amulet [Myers et al. 1996].

Approaches such as the above support temporal programming by introducing specific models and subsystems that relate only to time. The other possibility is that time is just another axis in a coordinate system, i.e., a t-axis in a coordinate system already containing an x-, y-, and perhaps a z-axis. This approach was used in an early spreadsheet language based on Lucid [Wadge and Ashcroft 1985] known as Plane Lucid [Du and Wadge 1990], which gave 2-D operators such as `hsby` and `vsby` for "horizontally/vertically spatially by (to the right/below)" to reference elements along the x- and y-axis of a spreadsheet grid and analogous operators such as `fby` meaning "followed by (temporally next)" to reference elements along the t-axis. The earlier Forms/3 approach discussed in the introduction was influenced by this work.

Gamut [McDaniel and Myers 1999] is a programming-by-demonstration VPL that also supports some temporal programming without introducing new models. Referencing the previous state of an object is done by inclusion of faded "temporal ghosts" in the demonstration area, so that they can be included in the demonstrated logic. The approach corresponds to our design goal of allowing a space-oriented programming technique to be applied to time, but does not allow time-oriented logic such as relative speed to be included, and allows reference only to the just-previous state, not to any other moment in time. The temporal ghosts device of Gamut is supplemented by a timer object device that relates only to time.

To summarize, previous VPL research that relates to temporal behavior can be divided into two categories. One is to provide entirely new language devices designed explicitly to support programming in time. The other is that time is just another dimension, able to be handled exactly like other dimensions. Our work falls in the latter category.

3. Grid-Based Space and Time

"Programming" (defining formulas for) spreadsheet grids has been mastered by millions of end users. This demonstrated ease of use motivates us to reapply this grid-based way of programming to time.

Here is how grid-based programming works in space. In the spreadsheet system Excel, the user puts a formula in one cell, and then replicates it (copies with the references automatically changed) across as much of the grid as desired. In the spreadsheet system Lotus, the user instead can group several cells in a grid and instruct the system that these cells all share the same formula. Forms/3 supports the latter strategy. In both strategies, the cells' formulas can refer to other cells by row and column number indices.

3.1 The Animated Sort Example

Software visualization is an important application of animated graphics, and thus we have chosen the animated sort as one of the motivating examples for this paper.

Since Forms/3 follows the spreadsheet paradigm, the group of elements being sorted is represented by a spreadsheet grid (also called a matrix in some papers). Prior to this paper, Forms/3's approach to time was similar to the stream-based approach found in many traditional languages. In Figure 2 from the introduction, there are 6 grids plus 1 standalone cell (`counter`). The overall logic is that `unsorted` starts as `input`, ("`input fby...`", where `fby`[2] means "followed by"), and then is either the result of removing the minimum element from its own earlier value or else is just a reference to its own earlier value. `sorted`'s logic is the mirror image of `unsorted`'s: it is either its own earlier value or else is the result of appending the minimum `unsorted` element to its own earlier value.

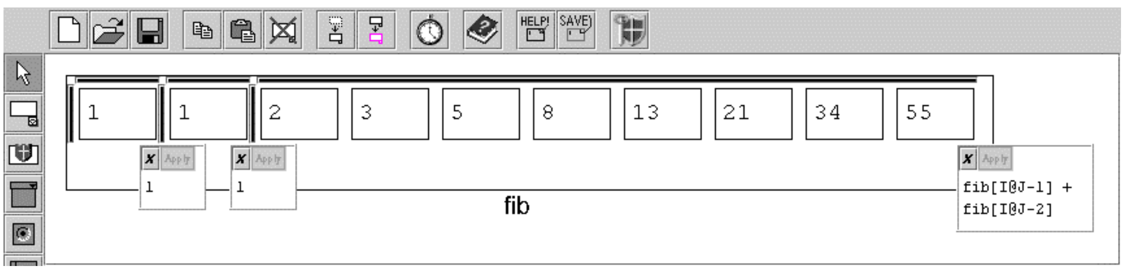


Figure 3: This Forms/3 spreadsheet contains a (spatial) grid of Fibonacci numbers. Note the thick border lines; these can be grabbed and dragged to divide a grid into regions. This grid contains 3 regions, and the cell values in each region are defined by the formula at the lower right of that region.

A user can subdivide a Forms/3 grid into regions by dragging a thick border line at the left of a grid rightwards or from the top downwards. All cells in a region share the same cell-level formula. (This is a simplified version of an approach pioneered by Viehstaedt and Ambler [Viehstaedt and Ambler 1992] and extended in Formulate [Ambler and Broman 1998, Wang and Ambler 1996].) Grid `input` was subdivided in this way, one cell per region so that each cell can have its own formula. See also Figure 3 for more examples of regions. Alternatively, a grid can have a formula that defines the entire grid as a whole, instead of the cells within it, as in `unsorted`, `prevSorted`, and `sorted`. (Grid `prevSorted` is present to break up what would otherwise have been a long, nested formula for grid `sorted`.)

Forms/3 grids have been empirically shown to be simpler for humans to use for some tasks than are more traditional methods [Pandey and Burnett 1993], but they do not seem particularly simple in Figure 2: much logic in this program is required for "slowing" down the program via cell `counter` so that the accompanying animation can perform smooth graphical transitions between element insertions into grid `sorted`. To accomplish this slow-down, most formulas contain `if`s that make decisions depending upon the status of `counter`. This is the kind of workaround necessary in languages that do not directly support the ability to program temporal relationships.

[2] `fby`, `if`, +, etc. are declarative formula operators. They define new values for the cells in which they reside whenever time moves.

The `fby` operator is typical of that used by many languages that explicitly support sequences over time. It exists explicitly in Lucid, Plane Lucid, and Chronolog [Liu and Orgun 1996], and is similar to functional list-construction operators, such as `cons` in Lisp, but is used to build lazy streams. The lazy stream approach has been used widely in the dataflow [Wadge and Ashcroft 1985], functional [Abelson and Sussman 1985], and reactive (also called synchronous) programming communities [Halbwachs 1993, Liu and Orgun 1996]. In these languages, streams are list-like structures that can, aided by special operators, lazily report computations over time. Higher-order styles such as Arya's functional animation processes [Arya 1989] and monads [Peyton Jones and Wadler 1993, Launchbury and Peyton Jones 1994] can be viewed as abstractions of the same concept.

While these approaches have been demonstrated to be sufficient for programming animated graphics, what we would like to do instead is to allow the specification of these temporal relationships without the use of traditional `fby`-like operators and without requiring workarounds to achieve varying speeds. Toward this end, we next consider several models of time to directly support temporal programming using only the grid concept.

3.2 A Continuum of Time Models

Several grid-based models of time are presented here in increasing order of power (functionality) but also of expected programming expertise. Given some model's placement along this continuum, we shall evaluate it by the number of problematic VPL design issues it introduces. The issues we consider are cognitive issues raised by Green/Petre's Cognitive Dimensions, and programming language design problems such as limitations and ambiguities. The models are:

1. Base Model: All grid cells have the same speed.
2. SNF Model: <u>S</u>low, <u>N</u>ormal and <u>F</u>ast. A user can choose one of a fixed number of speeds for a cell (e.g., `cellA` goes Fast). In our instantiation, the fixed number is 3: Slow, Normal, and Fast.
3. k*N Model: A user can specify a cell's speed as being an arbitrary integer constant <u>k</u> times faster/slower than <u>N</u>ormal (e.g., `cellA` goes 2 times faster than Normal).
4. k*x Model: A user can specify a cell's speed as an arbitrary integer constant <u>k</u> times faster/slower than the speed of another cell <u>x</u> (e.g., `cellA` goes 2 times faster than `cellB`).

4. Programming Along the T-Axis

4.1 The Base Model

Even without multiple speeds, temporal programming is useful for some purposes, one of which is specifying sequences. Figure 3 shows a (spatial) grid-based approach to programming the Fibonacci sequence, using the grid-oriented spatial relationships among the columns. The user separated this grid into the 3 regions shown by dragging the thick border line from the left of the grid rightwards. In Figure 4, the same technique was used to program a Fibonacci sequence over time instead of space, i.e., dragging the region separator border along the t-axis positions instead of the x-axis positions; this window is called the *temporal view* in Forms/3. As with grid-based programming, a formula can reference another cell's value using indices along an axis. In grid-based programming in Forms/3, x- and y-axis referencing is done via format `name[row@col]`; hence temporal referencing is done via format `name<time>`. For example, in Figure 4, `fib<t-1>` references `fib`'s value at the temporal position just before (to the left on the t-axis).

In principle, this is the same notion as that pioneered by Plane Lucid [Du and Wadge 1990], but the programming devices of our approach feature greater closeness of mapping to the domain of temporal programming. The relationships between formulas, ensuing values, and the t-axis are explicit in the way the user enters the formula (by directly moving borders among the values on the t-axis and pointing at other values to reference them) and in the fact that both the formula's temporal scope and values are explicit in their placement along the time line.

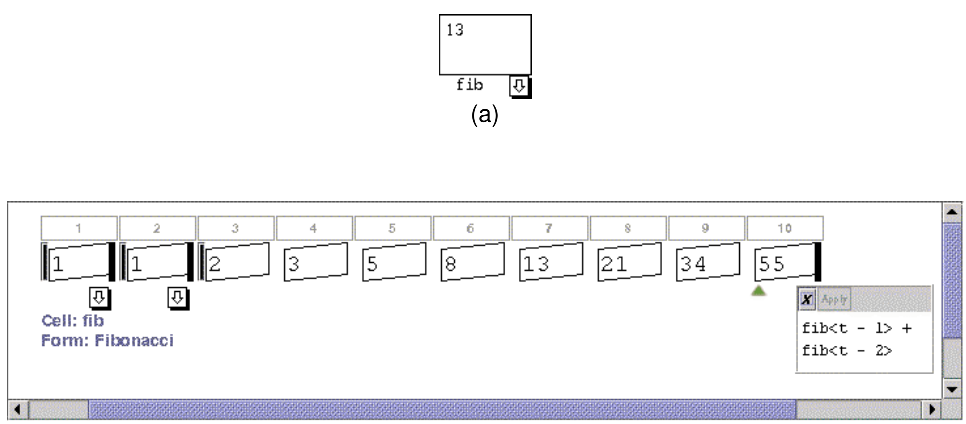


Figure 4: (a) This single Forms/3 cell can compute the Fibonacci sequence temporally under the base model; the computations are specified over time rather than space. (b) The temporal view of the same cell shows its 3 regions over time. Note the parallels to the spatial approach of Figure 3.

Because of the similarity to the spatial grid-based approach, users can reapply what they already know about programming in space to programming in time. In fact, the *only* thing this model does is add a new axis to the existing grid-based coordinate system. Every cell has exactly the same "logical" speed, i.e. is indexed along the t-axis in the same way as would be done on an x- or y-axis. Because every cell has the same speed, this model does not have the functionality to solve Problem 2, but it provides a basis upon which the later models can build.

Still, even this simple model brings out an intriguing issue about the difference between programming time and space. In space, it is usual for any grid item to be able to reference any other grid item, provided that no cycles are induced. But in time, the ability for the "past" to reference the "future" runs counter to the real world and thus could seem counterintuitive. We believe this flexibility could actually be used to advantage by an imaginative programmer, and would not be encountered by others. However, this is only our own opinion, and empirical work would be needed to determine the extent to which it actually caused problems for end users.

4.2 Model SNF: Slow, Normal, and Fast

The base model does not solve Problem 2, because all the cells have the same temporal speed. The goal of Model SNF is to solve Problem 2, for at least some cases, in a way usable by end users who know only grid-based programming.

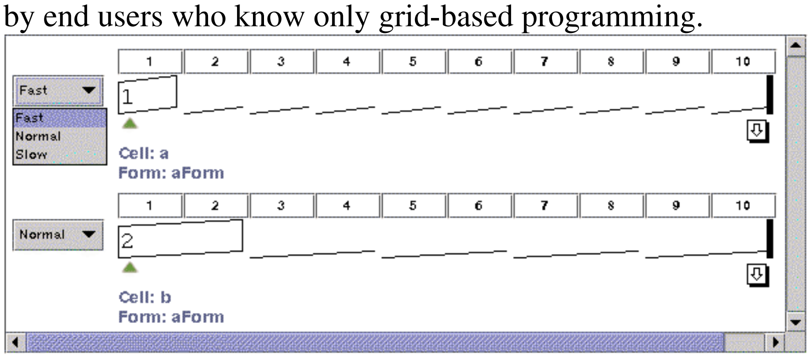


Figure 5: In Model SNF, a user specifies each cell's speed, supported here with a pop-up menu (e.g., top left) next to a cell in the temporal view. The small triangles point to the values at the current time.

In keeping with this goal, Model SNF adds to the base model by simply allowing the user to specify one of k speeds for a cell, where k is fixed in the language implementation. In our prototype, k=3, which allows speeds of Slow, Normal, and Fast. Also fixed in the language implementation are the relationships between Slow, Normal, and Fast. (We are considering allowing the user to modify this via a system preferences dialog.) In this discussion, we will set the speed difference factor to 2. If the speed difference were 10 instead using this model, the sorting algorithm's cells could have Normal speed (the default) and the user could set the speed of the animation cells to Fast, using the mechanism in Figure 5.

Allowing these multiple rates of speed raises the following question: what do temporal references such as Figure 4's <t-1> mean under this model? This question can be answered via the following three semantic rules:

Rule 1. The t-axis: As in the base model, in Model SNF there is one and only one time axis, it is known globally, and its indices are 1,2,… (These characteristics are the traditional characteristics of x- and y-axes.) We term each position along the t-axis a *tick*.

Ticks are not tied to “real” time units such as seconds or minutes; they are simply progressive positions on the t-axis. Speed Fast allows values at every tick, provided that dependencies in the formulas deliver enough data to populate every tick, speed Normal allows values at every 2 ticks, (1,3,...), and speed Slow allows values at every 4 ticks (1,5,...). See Figure 6.

In this and all the models in this paper, if there is a need to compute with or display a value at a time not explicitly populated for that cell, the most recent value before that time is still considered to be current. For example, if a slow cell’s value at time 3 is needed, its value at time 1 is considered still current and is used. A more conventional programming language view might be that this value should be considered to be undefined at times 2-4, but that is not the way time works in the real world. In the real world, slower moving objects do not have “undefined moments”; each intermediate state just lasts longer. This is visually depicted by the width of each value of slow cells, as in Figure 6.

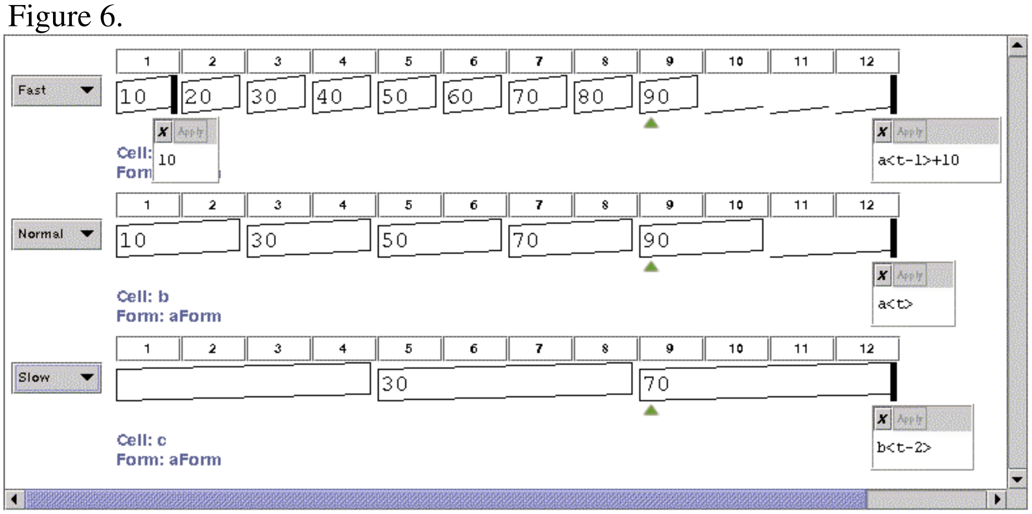


Figure 6: In Model SNF, cell values are located along a global t-axis as shown. Note that a cell is not restricted to only referencing cells of the same speed.

Rule 2. What t means: For time indices in formulas (e.g., b<t−2>), t is an index on the global t-axis that represents the current time.

Rule 3. What constants mean: For time indices in formulas (e.g., b<t−2>), constants follow the units of the global t-axis.

Intuitively, t means “now”. For example, if c refers to b<t−2>, then c’s value at time 5 will be the same as b’s value at time 3, as in Figure 6.

Rule 3 may seem puzzling: why is it necessary in a programming language to say what a constant means? The answer is that the multiple speeds introduce multiple unit possibilities. We will return to other unit possibilities later. An advantage to the unit choice in Rule 3 is that it explicitly relates to the t-axis indices visible in the VPL.

Under this set of rules, Model SNF adds the functionality needed to solve Problem 2, provided that only a fixed number of speeds are needed. Figure 7 and Figure 8 show how this model might be used by an end-user programmer to produce animated graphics. There are three x-position cells named `slow`, `normal`, and `fast` in this spreadsheet. They have similar formulas (increasing the previous element by 1 each step) but different speeds, and their values are numbers. The other three cells, `slowCrab`, `normalCrab`, and `fastCrab`, have values that are animated crabs moving to the right at different speeds; these values are defined as the value of cell `crab`—an image—at x-positions relative to the cells' origins equal to 8 times the values of cells `slow`, `normal`, and `fast`, respectively. (Like all operators in Forms/3, `compose` is a declarative operator that takes operands and produces a value—in this case, an image—that becomes the value of the cell. Since a user can put a cell anywhere desired, it is not necessary to do further calculations to translate local coordinates to a global system of coordinates [Burnett and Gottfried 1998].) Since the cells referred to (`slow`, `normal`, and `fast`) define new values as time advances, the three cells referring to them (`slowCrab`, `normalCrab`, and `fastCrab`), automatically produce new values whenever time advances.

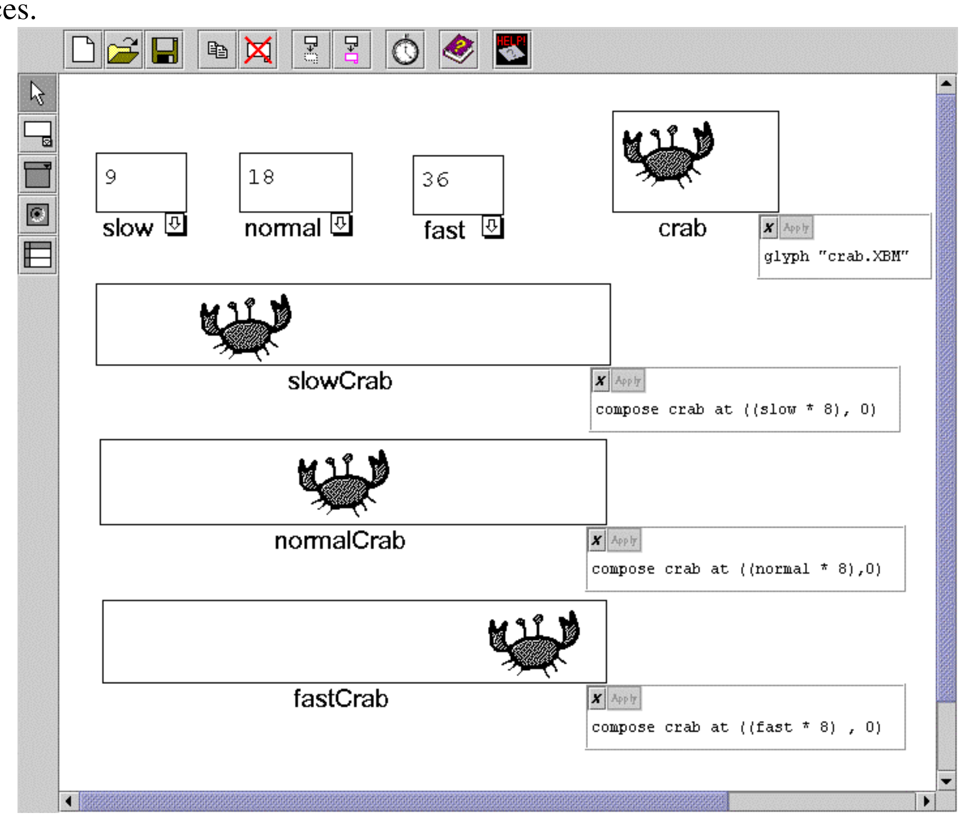


Figure 7: The `crabs` spreadsheet is an example of how Model SNF may be used by an end-user programmer. The formulas for cells `slow`, `normal` and `fast` are shown in Figure 8. The cells `slowCrab`, `normalCrab` and `fastCrab` compose the value of cell `crab` (which is an image) at different x,y positions over time (the y coordinate is always zero).

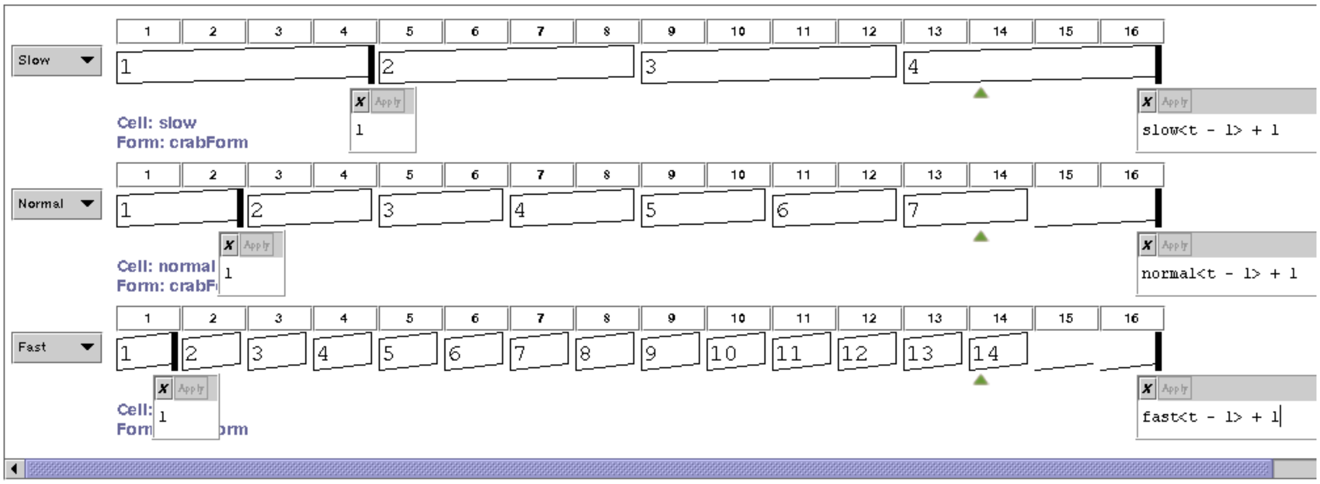


Figure 8: Temporal view of the three x-position cells in the `crabs` spreadsheet. (Since t refers to positions along the t-axis, cell `slow`'s formula could have been `slow<t-2>+1`, `slow<t-3>+1`, or `slow<t-4>+1`, with the same results, but these would be more fragile because their correctness relies on the cell's speed being set to "Slow".)

Despite the attractive simplicity of this model, it is not a panacea. In particular, the fixed number of speeds—which at first glance may seem only mildly limiting, at least for some applications—is quite problematic from the standpoint of two of the Cognitive Dimensions, premature commitment and viscosity. *Premature commitment* means users must make programming decisions before they have enough information to do so. In Model SNF, users must decide upon the speed of cells (or accept the default decision of Normal) even if they do not yet know if they will eventually want to add faster or slower cells related to it. Further, changing their minds later about a single cell's speed requires changing the speeds of, in the worst case, every cell in the program, in order to retain the desired temporal interrelationships. Thus *viscosity*, or resistance to change, is also a problem. Fortunately, removing the limitation on the number of speeds can help with these problems.

4.3 Model k*N: k Times Faster/Slower than Normal

Model k*N extends Model SNF such that any cell's speed can be specified as an arbitrary constant k times faster or slower than Normal; hence it removes the limitation from Model SNF.

Unfortunately, this straightforward extension leads to a new problem: what to do about the integer numbering specified in Rule 1. If Normal means 1 value every c ticks, then whenever the user selects a k > c, non-integer indices arise. For example, if Normal means 1 value per tick and a cell `a` were specified to be 3 times faster than Normal, it would need 3 values per tick, leading to t-axis numbering like (1, 1.333, 1.666, 2, 2.333, ...). Here the *consistency* Cognitive Dimension resurfaces as an issue, since this numbering system is clearly not consistent with the usual numbering of x- and y-axes.

One potential solution might seem to be to hide the unconventional t-axis numbering. However, hiding the indices would prevent users from knowing how to reference a previous element (e.g., <t-0.333>), so hiding the indices does not solve the problem.

A second potential solution might be to eliminate "faster than" as an option, allowing b for example only two speed choices: Normal or "k * slower than Normal". This retains integer numbering, but unfortunately also retains the premature commitment and viscosity problems of Model SNF, since now a commitment is needed that, if cell a is at Normal speed, no other cell will ever need to go faster than a.

A third potential solution is to retain integer numbering by automatically renumbering the t-axis indices when necessary. The problem here is that all the formulas relying upon a previous numbering would also have to be automatically changed. For example, consider cell c's formula of b<t-2> in Figure 6. If the user then added a new cell X 4 times faster than Normal—not referencing or being referenced by a, b, or c—she would probably be surprised if the system automatically changed c's formula to b<t-8>. Yet, making this change would be necessary: if the system did not make this change, c's formula would produce different answers than it had before. This is a classic case of *hidden dependencies*, the Cognitive Dimension that identifies dependencies that are not explicit in the program.

4.4 Model k*N Revised Via Rule 3

It is possible to leave Rule 1 unchanged, thereby avoiding these difficulties, by instead changing Rule 3, what constants mean. Until now, constants have been in t-axis ticks; e.g., <2> meant the 2nd tick, and <t-2> meant 2 ticks before t. However, another look at Figure 6 shows that there are multiple granularities along this axis. For example, in the figure, c's reference to b<t-2> at position 9 on the t-axis is b's value 2 *ticks* back, rather than b's value 2 *values* back. This presence of multiple granularities is significantly different from the way values populate x- and y- axes of grid-like systems. Changing Rule 3 allows the user to work directly in the granularity of values:

<u>*Rule 3 (revised). What constants mean:*</u> For time indices in formulas (e.g., b<t-2>), constants are mapped onto the units of the global t-axis by multiplying by the granularity of the *referenced* cell.

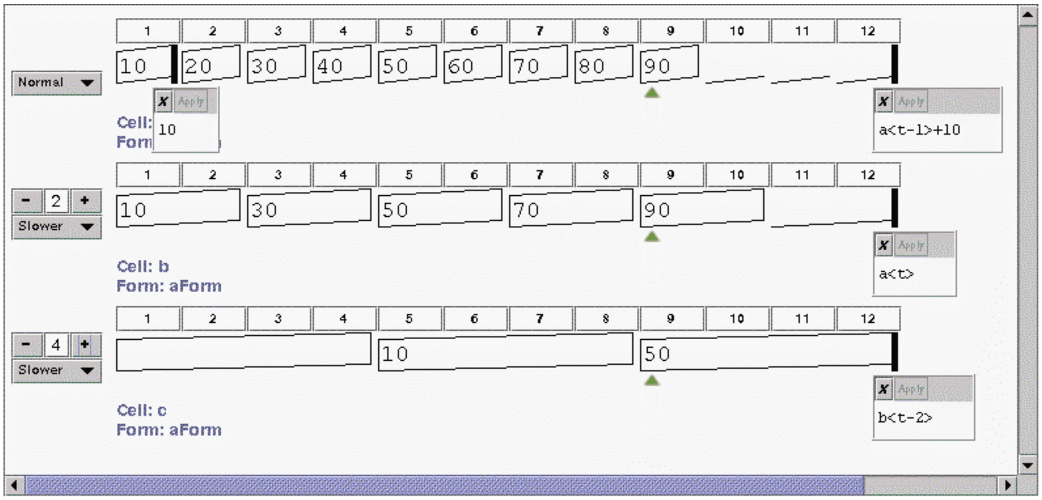


Figure 9: k*N Model revision via Rule 3.

For example, if `b`'s granularity is 2 times slower than the t-axis, populating the t-axis at t=1,3,5... as in Figure 6, then from the system's perspective, `c`'s reference to `b<t−2>` would refer to `b` at t=1-(2*2), 5-(2*2), 9-(2*2), ... as in Figure 9. (The first t-index in this sequence would be negative, resulting in no value for `c` until t=5.) Expressed from the user's perspective, `c`'s reference to `b<t−2>` refers to `b` two *values* before now.

With this revision the t-axis can be renumbered whenever doing so is necessary to retain integer numbering of the t-axis. The renumbering will not require any formulas to be changed, and the user will not have to perform the above granularity conversions manually that would have arisen with multiple speeds.

The revision to Rule 3 might seem to threaten referential transparency (the ability to substitute equals for equals), since it causes the meaning of constants to vary contextually. In Figure 9, if cell `c`'s formula were "`a<t−2> + b<t−2>`", then `c`'s "`a<t−2>`" value at time 9 would be cell `a`'s value at time 7 (`a<9−2*1>=a<7>`). However, `c`'s value for "`b<t−2>`" would be cell `b`'s value at time 5 (`b<9−2*2>=b<5>`). Thus, in this example, "`<t−2>`" means different time positions, based on which cell is referenced. But as long as the referenced cell and its time referencing specification are treated as an atomic unit, referential transparency is preserved.

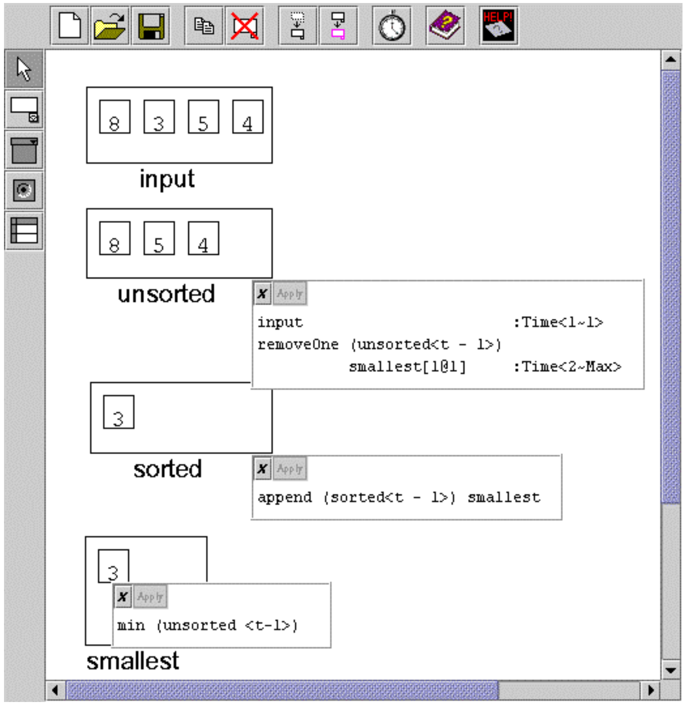


Figure 10: The selection sort under Model k*N. Compare this figure to Figure 2. In this view, `unsorted` shows the formulas of all the different time regions together; the notation `:Time<a~b>` next to an expression defines where a time region starts, where it ends, and which expression is used for that region. See Figure 11 for a temporal view.

This model has a compelling advantage: it avoids or reduces many of the difficulties of previous potential solutions and solves both Problems 1 and 2. Figure 10 demonstrates this with the new version of the sort[3]. Compare this figure to Figure 2: all the former `if`'s devoted to slowing down the timing are gone, and now only 3 formulas, all of which are now non-nested, remain in the sorting program: one to select the `smallest` unsorted element, one to remove it from the `unsorted` grid, and one to append it to the `sorted` grid. These grids' speeds are specified as Normal, and the animation cells' speeds are all specified as 10 times faster than Normal.

[3] With different time indexing (e.g., `t-10` instead of `t-1`), this solution is also possible in Model SNF, but only if the speed differential between speeds were permanently set to 10 in the language implementation.

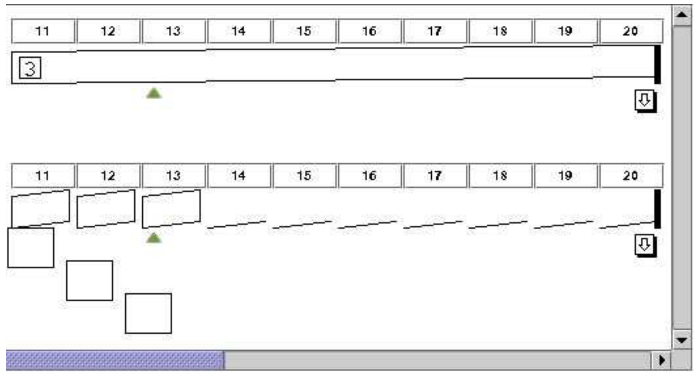


Figure 11: Temporal view of the `sorted` grid (top row) and the cell containing the rectangular box value moving downward through different positions 10 times faster than normal (bottom row). The box moves over time in a way similar to the crabs described in Section 4.2, using the `compose` operator.

Figure 11 shows the temporal view of the `sorted` grid and the relevant subcell in the `sortOutput` grid. The second cell's speed is 10 times faster than the `sorted` grid. Initially there is no value in the `sorted` grid; thus the corresponding subcell of the `sortOutput` grid has a stationary box from time 1 to time 10. At time 11, when the value 3 appears in the `sorted` grid, the box begins to move.

There are disadvantages to this model, but they are not nearly as troublesome as the problems encountered prior to this revision. One disadvantage is that this variant of Rule 3 is less explicitly tied to the visible t-axis numbering. The loss of explicitness is because the subtraction of 2 from t in `<t-2>` in Figure 9 is in different units than before: number of values rather than number of ticks. (An alternative would be to remove all ties to the t-axis, which we briefly discuss in Section 6.) Premature commitment and viscosity are also still issues in this model, although to a lesser extent than in Model SNF, because if two cells' speeds exist at adjacent integer factors (e.g., 2 times faster than Normal and 3 times faster than Normal), it is not possible to insert a new speed between those two.

4.5 Model k*x: k Times Faster/Slower than x

Model k*x is such a straightforward extension of Model k*N, few words are required to describe it; yet, it brings highly useful properties. The only difference from Model k*N is that, instead of specifying a cell's speed to be k times faster than Normal as in Model k*N, the user specifies the cell's speed to be k times faster than some other cell `x`. The revision of the k*N model should be applied to this model too: revising the k*x model via Rule 3 brings the same benefits to the k*x model as to the k*N model.

Thus, the spreadsheet of Figure 10 is exactly the same as that needed under this model. The temporal view would be almost the same under this model, with the small change that, in the speed setting for the animation cell, the user would specify that it was 10 times faster *than* `sorted`.

Model k*x features better closeness of mapping than the previous model, because the user can specify the temporal relationships among cells directly, instead of translating them all into temporal relationships with Normal speed. Further, this model under the same Rule 3 revision as in Section 4.4 removes the problems of premature commitment and viscosity while retaining the advantages of solving Problems 1 and 2.

However, with the improvement of closeness of mapping, the problem of hidden dependencies re-emerges to some extent. In previous time models, all cells' speeds are dependent on a fixed set of speeds (Slow, Normal, or Fast). The user can explicitly see how fast a cell goes by looking at its speed selection on the left of the temporal view. However, in this time model, a cell's speed can depend on any other cell's speed. Because of this, although the dependencies "Whose speeds are my speed dependent on?" are explicit, the dependencies "Whose speeds are dependent on me?" are hidden. Thus, if a user changes one cell's speed, he or she cannot directly view what other cells' speeds dependent on this cell's speed will be changed too.

5. An Empirical Study

Throughout this paper, we have looked to the Cognitive Dimensions to find potential problems related to end users' use of these language devices. However, as mentioned before, Cognitive Dimensions can help find problems but cannot actually "prove" usability.

To find out if any of these models are actually usable by end-user programmers, we conducted an empirical study. We chose Model SNF as the grid-based one to study because its initial learning time was expected to be small, an important consideration given the limited time possible for initial learning in a lab setting. We chose the traditional stream-based approach with temporal operators such as `fby` to compare it with because it is representative of the approach used in many other languages, and in this sense represents a "standard" in temporal programming.

The objectives of our study were to investigate the following research questions:

RQ1: Do end users who are given Model SNF to express temporal relationships modify a spreadsheet more correctly than end users who are given the traditional approach?

RQ2: Do end users who are given Model SNF to express temporal relationships modify a spreadsheet faster than end users who are given the traditional approach?

RQ3: Do end users have better understanding of a spreadsheet that uses the Model SNF than a spreadsheet that uses the traditional approach?

To answer these research questions we first asked the participants to modify temporal relationships in a spreadsheet using a PC. Then we asked them to fill in values on a second spreadsheet without using the computer. Half of the participants completed the experiment using the Model SNF approach and the other half used the traditional approach.

The study was conducted one participant at a time. First, the lecturer led a hands-on tutorial and practice session in Forms/3, introducing the participants to the concept of temporal relationships and how they can be expressed in the Forms/3 environment. Half of the participants were given this instruction only for Model SNF, and the other half were given this instruction only for the traditional approach. Participants in both groups received the same time for training and practice. The kinds of modifications to the spreadsheet taught during the tutorial were also identical, regardless of which time model they were using. To minimize short-term memorization skill differences as a determining factor, participants were also given a Forms/3 reference card to refer to as needed throughout the experiment.

Following the tutorial, the participants were asked to modify a spreadsheet using the computer and then to answer a written test designed to measure the understanding of a second spreadsheet without using the computer. These tasks will be described in Section 5.2. The data produced by the study included pre-task and post-task questionnaires, electronic transcripts of participants' on-line activities, their final modified spreadsheets, and their written comprehension tests.

5.1 The Participants

Our participants were drawn from a computer literacy course. The participants were randomly divided into the two groups. Two weeks before the experiment they had completed the spreadsheet module in their class. Thus, all of the participants had previous experience using the Excel spreadsheet system; two also had previous experience with other spreadsheets. None of the participants had used Forms/3 before. In our background questionnaires, none of the participants claimed to be an expert or a heavy user of spreadsheets and none of the participants had programming experience. Table 1 shows the previous spreadsheet experience of the participants.

Group	Excel	Lotus	Other	Forms/3
FBY	16	0	1	0
SNF	16	1	0	0

Table 1: Previous spreadsheet experience of the end users participating in the empirical study.

5.2 The Problems

The Crabs spreadsheet. This spreadsheet, based on the one described earlier in Section 4.2, produces animated graphical output. Recall that the three crabs move along the x-axis. In the task involving this spreadsheet, the participants were to modify the speeds for the cells `slow`, `normal`, and `fast` (Figure 12); these cells define the position on the x-axis of the crabs in cells `slowCrab`, `normalCrab`, and `fastCrab` respectively. As given to the participants, the crabs were set to move at the same pace. The participants' goal was to make `normalCrab` move twice as fast as `slowCrab`, and `fastCrab` move twice as fast as `normalCrab`. We chose this problem because a primary motivation behind our work is to allow spreadsheet users to program animated graphics.

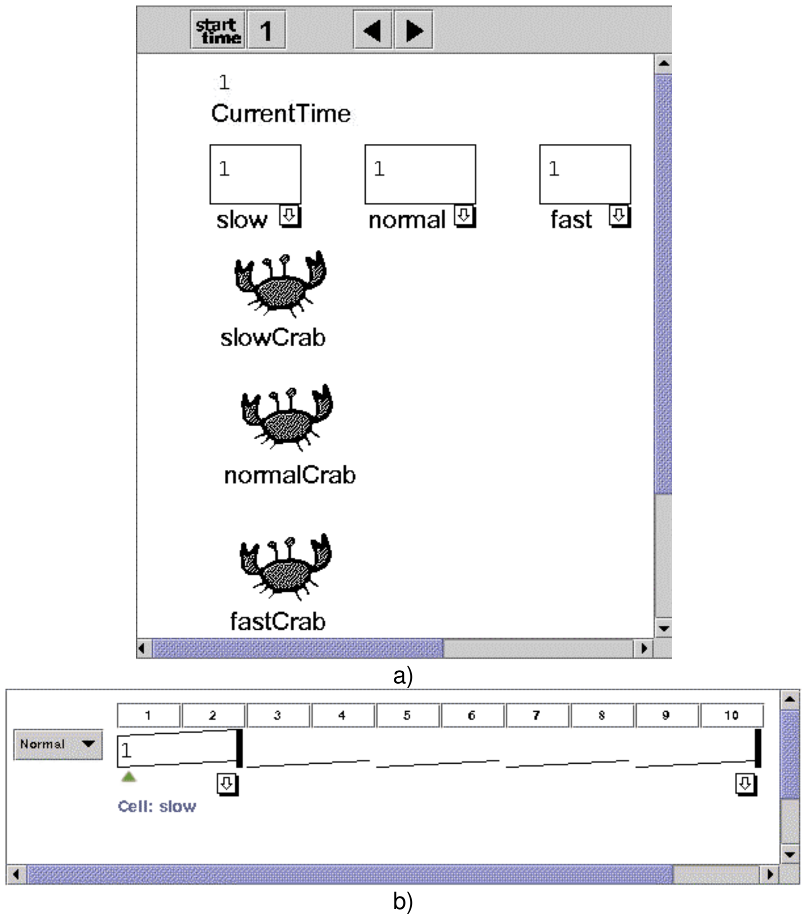
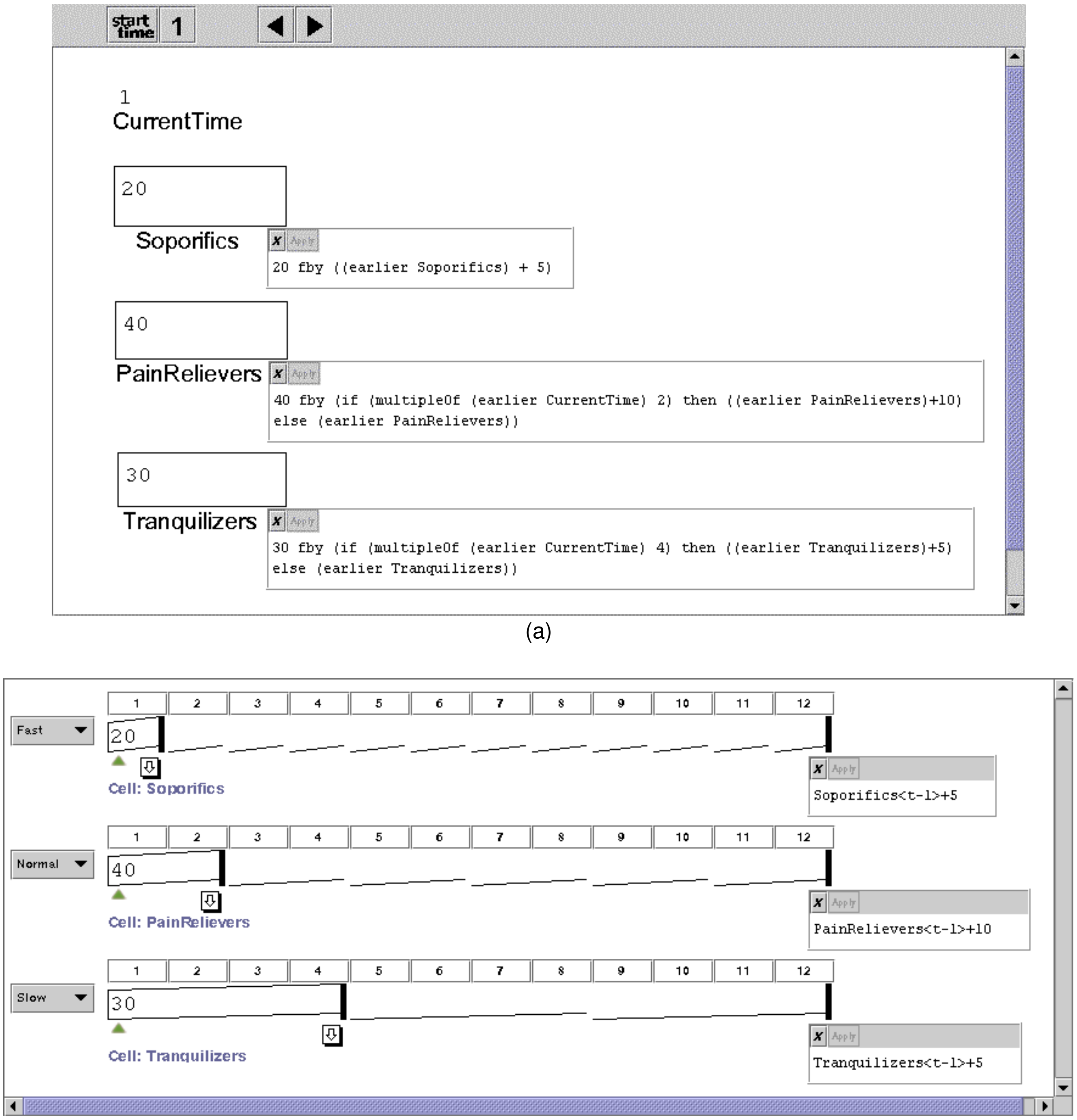


Figure 12: a) The `crabs` spreadsheet as initially presented to participants. b) An example of the temporal view window that participants in the SNF group could use to modify the speed of cells. Initially all cells had speeds set to Normal.



(b)

Figure 13: (a) Spreadsheet seen by participants in the traditional group. (b) Spreadsheet seen by participants in the SNF group.

The Drug Company Deliveries problem. This spreadsheet was used in the written comprehension test. It is based on the spreadsheet used in [Saariluoma and Sajaniemi 1994]. In that paper the participants were asked to act as bookkeepers for a fictitious medical company. Their task was to find information for executives about possible errors in deliveries by using a single formula. In our experiment, we added a time factor. The spreadsheet represents the deliveries of three different pharmaceutical products. Participants were given a picture of the spreadsheet and were asked to figure out how much the company will deliver for each of the three products in the 5th and 8th months.

Then they were given the same spreadsheet with modified formulas/speeds and asked to answer the same questions again. See Figure 13.

5.3 RQ1: Correctness Results

Our first research question considers whether using Model SNF to express temporal relationships improved the correctness with which the participants could complete their programming tasks. A summary of the results is shown in Figure 14. To measure the correctness of the spreadsheets modified by the participants we scored the correctness of the final formulas using the following criteria: if the position of a crab was the same as we asked, we gave one point; if the crab was not at the specified position but was closer to the desired position than it had been initially, we gave one half of a point. If the crab was closer to the desired position than it had been initially but the participant altered the initial position for the crabs, we gave one fourth of a point. The maximum possible score was 3 (one point for each crab).

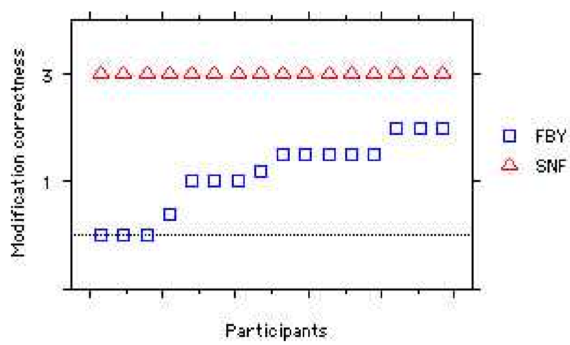


Figure 14: Participants in the SNF group modified the Crabs spreadsheet more correctly than participants in the fby group (SNF mean = 3.0, fby mean = 1.13). In the graph the participants are sorted according to their performance.

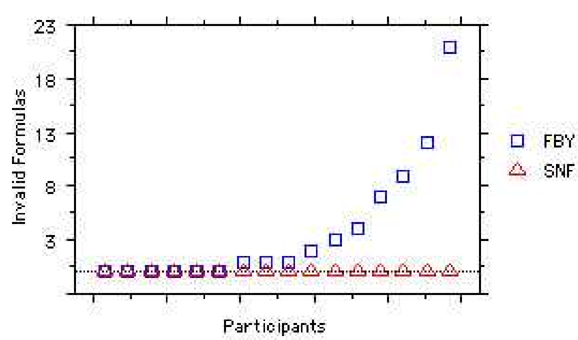


Figure 15: No participants in the SNF group entered any syntactically invalid formulas even temporarily while modifying the Crabs spreadsheet (they are shown over the dotted line), whereas participants in the fby group entered an average of 3.82 invalid formulas (some of which were eventually corrected).

As another way of detecting difficulties participants might be experiencing, we also measured how many syntactically invalid formulas the user entered along the way, regardless of whether they were ultimately corrected. See Figure 15. While SNF participants did not have opportunities to make syntax errors on the speeds per se, some of them also modified the other parts of the formulas, and there it was possible to make syntax errors. Since dealing with syntax errors can turn a user's attention away from the logic of a problem's solution, the differences in activity spent on syntax may at least partially explain the differences in the correctness results.

As the figures show, the differences were very large. Our analysis of the data showed that these differences were statistically significant. Significantly more participants in the traditional group entered invalid formulas along the way (Fisher's Exact test, $p = .0002$) and formulas of participants in the SNF group were significantly more correct (Mann-Whitney test, $p < .0001$).

5.4 RQ2: Speed Results

Our second research question considers whether participants using Model SNF would modify temporal relationships faster than those using the traditional approach. Using the electronic transcripts of the on-line activity of our participants, we measured the elapsed time between the first interaction with the spreadsheet (showing a formula, editing a formula, changing the speed of a cell, etc.) and the last action. See Figure 16. The participants had 15 minutes to complete the task, but some of the participants in the fby group were allowed to use a couple more minutes if they felt they were close to the solution. However, in our statistics scores in excess of 15 minutes were scored as 15 minutes. Thus, since this allowance favored the traditional group's correctness, and did not affect the statistics on speed, it does not invalidate either the correctness or the speed results.

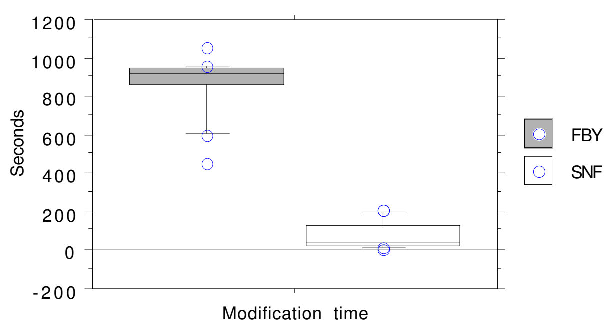


Figure 16: Time to perform modification of Crabs spreadsheet. The box plots are composed of 5 horizontal lines at the 10th, 25th, 50th, 75th and 90th percentiles; values above the 90th or below the 10th percentiles are plotted separately. Note that all the SNF participants were faster than *any* fby participant.

Participants using Model SNF modified the Crabs spreadsheet significantly faster than participants using the traditional approach (Mann-Whitney test, $p < .0001$). In fact, the *slowest* participant using the SNF approach spent only three minutes and forty three seconds getting all the modifications right, whereas the *fastest* participant using the traditional approach spent seven minutes and twenty eight seconds and did not get any modifications right.

As another measure of speed, we also counted the number of edits the participants performed. As Figure 17 shows, participants in the SNF group performed fewer edits in the Crabs spreadsheet than participants using the traditional approach (Mann-Whitney test, p<.0001).

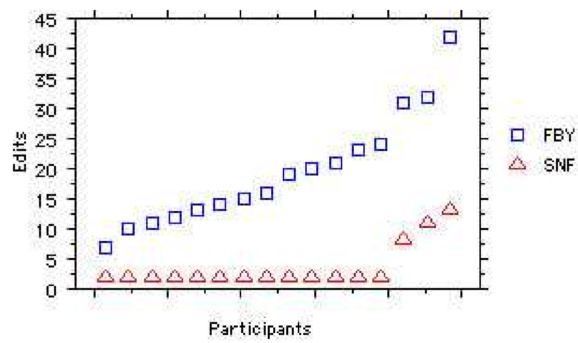


Figure 17: Number of edits performed to modify the Crabs spreadsheet (SNF mean=3.62, fby mean=19.37).

5.5 RQ3: Understandability results

Our last research question focuses on ability to understand programs written by someone else. To measure understandability we used the Drug Company Deliveries problem described in Section 5.2. The test consisted of 12 questions and we awarded 1 point for each correct answer. Thus, the maximum score possible in the test was 12. Participants in the SNF group scored significantly higher in this task (Mann-Whitney test, p=.0003). See Figure 18.

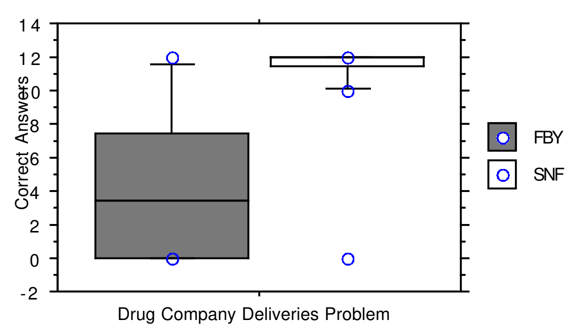
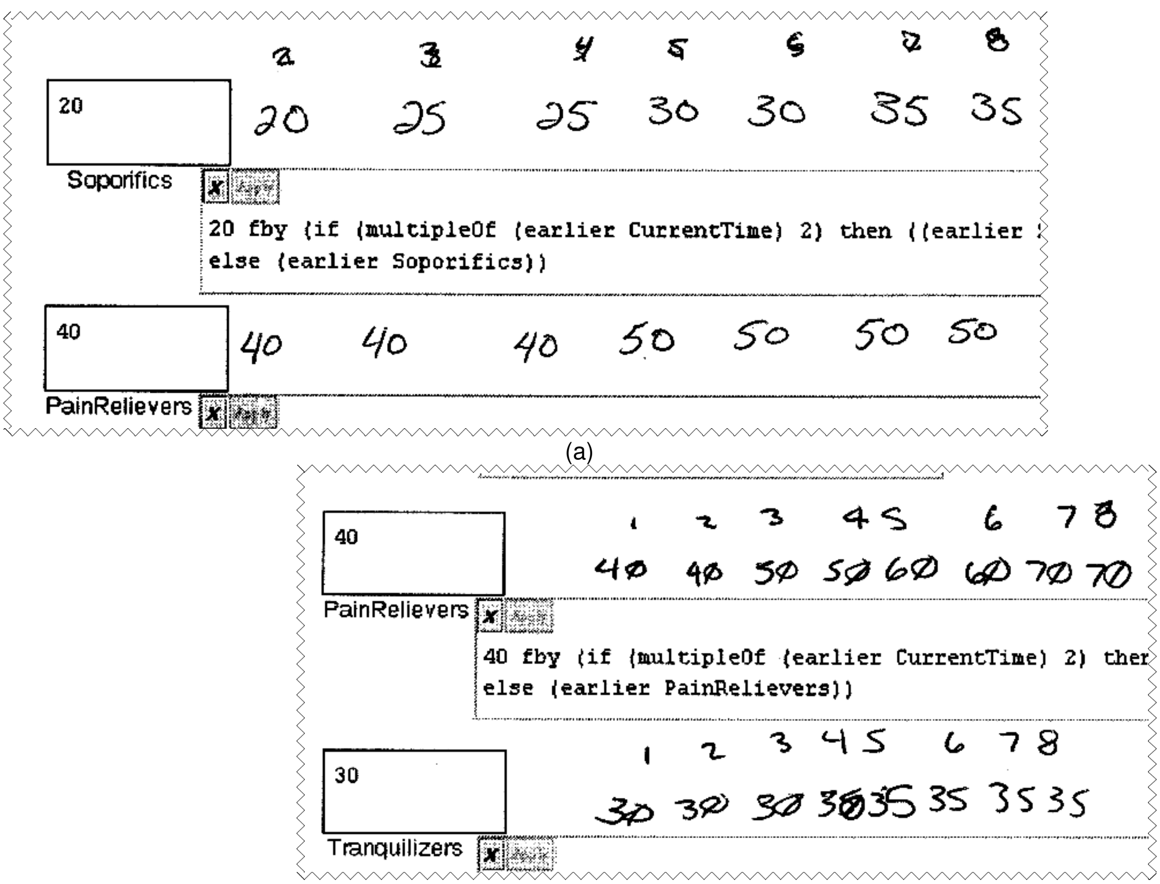


Figure 18: Number of correct answers in the Drug Company Deliveries problem (SNF mean = 11, fby mean = 4.31).

Two participants using the traditional approach got all of the answers correct in this task. We were very interested to see that both of these participants supported their answers with a drawing very similar to the temporal window used by the participants in the SNF group. Recall that the traditional group participants were never shown temporal windows. This suggests that the grid-based temporal window is a close match to the way some end users choose to think about this kind of problem. See Figure 19.

5.6 Discussion

Recall that the Cognitive Dimensions of premature commitment and viscosity were revealed as two drawbacks of Model SNF. This was not contradicted by our empirical results; rather, these issues generally do not have time to arise in a short controlled experiment. In particular, in the crab spreadsheet problem, the participants will not have to deal with future modifications, and thus are not faced with premature commitment or with viscosity. This seeming difference in usability results simply demonstrates the fact that no single evaluation technique can alone reveal complete usability information.



(a)

(b)

Figure 19: The two fby participants who correctly answered all the questions in the second task supported their answers with drawings that closely resemble the window used by our visual models. (a) and (b) are scanned-in drawings these two participants wrote on their test forms.

As mentioned earlier, the traditional approach is used in several languages. It is representative of the one-dimensional, stream-based approaches and therefore presents a standard for comparison. Still, we tried to simplify the knowledge needed to use the traditional approach so that results would depend on the approach itself, not extraneous factors. Toward this end, we added an `isMultipleOf` operator, so that the participants did not need to use the `mod` operator, which our pilot revealed to be unfamiliar to some non-programmers.

Recall that it is necessary in the traditional approach to use conditions (`if` operators) to decide if a given cell should be calculated at the current time, as was demonstrated earlier in the animated sort. In the electronic transcripts we observed that these conditional expressions were a particular source for erroneous formulas. Since the participants had no programming experience, it is not surprising that they had trouble with `if` expressions, even though they were given tutorial instructions and practice, and even though they had a reference card throughout the experiment explaining the spreadsheet operators.

Given the strong understandability results for the SNF approach, a natural question that arises is whether some particular portion of the approach was key to the results. Although we do not have a rigorous answer to this question, we have the opinions of the participants. In the post-task questionnaires we asked the SNF participants "Did the temporal window help you check that your modifications were correct?" Thirteen out of the sixteen SNF participants answered "Yes" to this question, and the other three participants answered "No." Of the participants who answered affirmatively, eleven said in the unstructured "Why?" follow-up question that they liked the fact that they were able to visualize how the value of the cells changed over time.

The results of the empirical study are very strong, but may not generalize to the other models presented in this paper, other problems, other populations of users, or longer-term tasks. Further empirical studies are needed to determine these issues.

6. Future Work

The models presented are all static in the sense that the cell's temporal speed can be statically determined without the need for run-time data. We are also considering experimenting with an inherently dynamic model, a y*x model, in which a user can specify a cell's speed as an arbitrary computation <u>y</u> faster/slower than the speed of another cell <u>x</u> (e.g., `cellA` goes `cellB` times faster than `cellC`). Since `y` is an expression that could be a cell reference (e.g., `cellB`), its formula could be arbitrarily complex, and could itself vary over time, allowing special effects such as "slow-in slow-out", speeds computed by functions, speed maximums, and so on. Also, under this model (and under Model k*x), it is possible to refer to a distinguished cell, `System:clock`, which reports the time of day in seconds (e.g., `cellA` goes `cellB` times slower than `System:clock`). This supports the capability of controlling speed in terms of elapsed seconds, although we have not yet experimented with this to explore its limitations.

A more fundamental addition we are working on is to bring some of the elements of computer graphics to bear on temporal programming, such as by employing local coordinate systems and operations such as “scale”. However, these changes rest upon a change in Rule 2 (what t means) to a local notion of t, and this in essence completely separates the global t-axis from any use of t in a formula. An implication of this separation is that it makes the numbering of the t-axis useless, which shows how far removed this approach is from grid-oriented x- and y-axis programming. Pursuing this approach could yield an approach in which temporal programming can be done as a simple extension to spatial programming, but the model of spatial programming it extends is graphics programming (normally done by professional programmers) as opposed to grid-based programming (often done by end users). Thus this new model will probably be best suited to people with some programming background; we do not expect it to be comfortable to end users with no programming background.

7. Conclusion

Although several prior researchers have made the point that temporal programming should be achievable using the same mechanisms as spatial programming, this possibility has not been thoroughly investigated. In this paper, we have added to what is known about this possibility for grid-based VPLs by attempting to solve two specific problems in this way. The motivation is this: end users succeed at grid-based programming in space, which suggests that an approach consistent with this for programming temporal behavior would allow this audience to reapply what they already know to temporal programming applications such as animated graphics.

The models we have developed reveal several core differences between the ways grid-based programming have been supported in space and the extension of such ways to temporal programming:

- The ability to access *any* position on the x- or y-axis loses its consistency with the real world on the t-axis: what does it mean for a past position to reference the future?
- In the real world, in space, if a value is not present at some specific position then it is missing, whereas in time, a situation at time t persists until something different occurs. Upholding these real-world conventions in a VPL seems necessary and useful but introduces inconsistency between space and time.
- Whereas spatial grid element sizes are constant in the grids found in the VPLs of which we are aware, multiple granularities on the t-axis are needed to support varying speeds.

These differences did not lead to significant problems in the first model (Model Base), but because that model does not accommodate the third difference, it does not solve Problem 2. Accommodating the differences, especially the third one, in the next two models introduced several problems such as premature commitment, viscosity, lack of consistency, and hidden dependencies, all of which have been shown to cause difficulties for humans. The last model (Model k*x) overcomes these problems to a large degree, and supports direct, grid-based specification of an unlimited number of speeds and temporal relationships, solving both Problems 1 and 2. However, also recall that the models were presented along a continuum of expected programming expertise required, and Model k*x is expected to require more programming expertise than the others.

In our empirical results, end users with light to moderate spreadsheet experience and no programming experience were able to succeed at both programming and at understanding temporal behaviors in one of these models, Model SNF. Further, the results showed that their ability to succeed was significantly greater than in the traditional approach, which most were not able to use successfully. These results strongly suggest that the traditional approach is not a reasonable language feature for this population, but that a grid-based model is a viable approach. However, these results were specific to Model SNF. Further empirical investigation is required to determine the level of spreadsheet expertise required for mastery of the k*N and k*x models.

Acknowledgments

We thank the members of the Visual Programming Research Group at Oregon State University for their feedback and help with the implementation. We also thank Thomas Green for feedback on the experiment's design, Mojgan Hajebi for allowing us to promote the experiment to her students, and the students in her Fall 2000 class for their participation. Special thanks go to Rosana Duran, Ismael Garcia and Caryn Jensen for their help testing the empirical study. This work was supported in part by the National Science Foundation under awards CCR-9806821 and ITR-0082265.

References

[Abelson and Sussman 1985] H. Abelson and G. Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, Massachusetts, 1985.

[Ambler and Broman 1998] A. Ambler and A. Broman, Formulate solution to the Visual Programming Challenge, *Journal of Visual Languages and Computing* 9(2), 171-209, Apr. 1998.

[Arya 1989] K. Arya, Processes in a functional animation system, *Functional Programming and Computer Architecture*, London, 382-395, Sept. 1989.

[Atwood et al. 1996] J. Atwood, M. Burnett, R. Walpole, E. Wilcox, and S. Yang, Steering programs via time travel, *IEEE Symposium on Visual Languages*, Boulder, Colorado, 4-11, Sept. 1996.

[Brown 1988] M. Brown, *Algorithm Animation, An ACM Distinguished Dissertation 1987*, MIT Press, 1988.

[Burnett and Gottfried 1998] M. Burnett and H. Gottfried, Graphical definitions: Expanding spreadsheet languages through direct manipulation and gestures, *ACM Transactions on Computer-Human Interaction* 5(1), 1-33, Mar. 1998.

[Burnett et al. 1998] M. Burnett, J. Atwood, Z. Welch, Implementing level-4 liveness in declarative visual programming languages, *IEEE Symposium on Visual Languages*, Halifax, Canada, Sept. 1998.

[Burnett et al. 2000] M. Burnett, N. Cao, J. Atwood. Time in grid-oriented VPLs: Just another dimension? *IEEE Symposium on Visual Language*s, Seattle, Washington, 137-144, Sept. 2000.

[Burnett et al. 2001] M. Burnett, J. Atwood, R. Djang, H. Gottfried, J. Reichwein, and S. Yang, Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm, *Journal of Functional Programming*11(2), 155-206, Mar. 2001.

[Carlson et al. 1996] P. Carlson, M. Burnett, and J. Cadiz, A seamless integration of algorithm animation into a visual programming language, *ACM Proceedings of the Workshop on Advanced Visual Interfaces*, Gubbio, Italy, 194-202, May 1996.

[Chi et al. 1998] E. Chi, J. Riedl, P. Barry, and J. Konstan, Principles for information visualization spreadsheets, *IEEE Computer Graphics and Applications*, July/Aug. 1998.

[Du and Wadge 1990] W. Du and W. Wadge, A 3d spreadsheet based on intensional logic, *IEEE Software*, 78-89, May 1990.

[Duecker et al. 1997] M. Duecker, C. Geiger, R. Hunstock, G. Lehrenfeld, W. Mueller, Visual-textual prototyping of 4d scenes, *IEEE Symposium on Visual Languages*, Capri, Italy, 328-335, Sept. 1997.

[Duisberg 1986] R. Duisberg, Animated graphical interfaces using temporal constraints, *ACM Conference on Human Factors in Computer Systems (CHI'86)*, Boston, Massachusetts, 131-136, Apr. 1986.

[Duisberg 1990] R. Duisberg, Visual programming of program visualizations: A gestural interface for animating algorithms, in *Visual Languages and Applications* (T. Ichikawa, E. Jungert, R. Korfhage, eds.), Plenum Publishing, New York, 1990.

[Freeman et al. 1996] E. Freeman, D. Gelernter, and S. Jagannathan, Uniformity of environment and computation in MAP, *IEEE Symposium on Visual Languages*, Boulder, Colorado, 130-137, Sept. 1996.

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

[Halbwachs 1993] N. Halbwachs, *Synchronous Programming of Reactive Systems*, Kluwer, 1993.

[Heger et al. 1998] N. Heger, A. Cypher, and D. Smith, Cocoa at the Visual Programming Challenge 1997, *Journal of Visual Languages and Computing* 9(2), 151-169, Apr. 1998.

[Hibino and Rundersteiner 1997] S. Hibino and E. Rundersteiner, User interface evaluation of a direct manipulation temporal visual query language, *ACM International Multimedia Conference*, Seattle, Washington, 99-107, Nov. 1997.

[Ioannidou and Repenning 1999] A. Ioannidou and A. Repenning, End-user programmable simulations, *Dr. Dobb's Journal*, 40-48, Aug. 1999.

[Kutty et al. 1993] G. Kutty, L. Dillon, L. Moser, P. Melliar-Smith, and Y. Ramakrishna, Visual tools for temporal reasoning, *IEEE Symposium on Visual Languages*, Bergen, Norway, 152-159, Aug 1993.

[Launchbury and Peyton Jones 1994] J. Launchbury and S. Peyton Jones, Lazy functional state threads, *ACM Conference on Programming Language Design and Implementation*, June 1994.

[Levoy 1994] M. Levoy, Spreadsheet for images, *ACM Siggraph 94*, 139-146, 1994.

[Liu and Orgun 1996] C. Liu and M. Orgun, Dealing with multiple granularity of time in temporal logic programming, *Journal of Symbolic Computation* 22, 699-720, 1996.

[MacNeil 1991] R. MacNeil, Generating multimedia presentations automatically using TYRO, the constraint, case-based designer's apprentice, *IEEE Workshop on Visual Languages*, Kobe, Japan, 74-79, Oct. 1991.

[McDaniel and Myers 1999] R. McDaniel and B. Myers, Getting more out of programming-by-demonstration, *ACM Conference on Human Factors in Computing Systems*, Pittsburgh, Pennsylvania, 442-449, May 1999.

[Myers et al. 1996] B. Myers, R. Miller, R. McDaniel, A. Ferrency, Easily adding animations to interfaces using constraints, *ACM Symposium on User Interface Software and Technology*, Seattle, Washington, 119-128, Nov. 6-8, 1996.

[Pandey and Burnett 1993] R. Pandey and M. Burnett, Is it easier to write matrix manipulation programs visually or textually? An empirical study, *IEEE Symposium on Visual Languages*, Bergen, Norway, 344-351, Aug. 1993.

[Peyton Jones and Wadler 1993] S. Peyton Jones and P. Wadler, Imperative functional programming, *ACM Symposium on Principles of Programming Languages*, Charleston, South Carolina, 71-84, Jan. 1993.

[Saariluoma and Sajaniemi 1994] P. Saariluoma and J. Sajaniemi, Transforming verbal descriptions into mathematical formulas in spreadsheet calculation, *International Journal of Human-Computer Studies* 41(6), 915-948, 1994

[Song et al. 1996] J. Song, M. Kim, and G. Ramalingam, Interactive authoring of multimedia documents, *IEEE Symposium on Visual Languages*, Boulder, Colorado, 276-283, Sept. 1996.

[Stankovic and Zhang 1997] N. Stankovic and K. Zhang, Towards visual development of message-passing programs, *IEEE Symposium on Visual Languages*, Capri, Italy, 144-151, Sept. 1997.

[Stasko 1989] J. Stasko, *TANGO: A Framework and System for Algorithm Animation*, Ph.D. Dissertation, Tech. Rpt. CS-89-30, Brown University Dept. of Computer Science, May 1989. Summarized under same title in *IEEE Computer* 23(9), 27-39, September 1990.

[Stasko 1991] J. Stasko, Using direct manipulation to build algorithm animations by demonstration, *ACM Conference on Human Factors in Computing Systems*, , New Orleans, Apr. 27 - May 2, 1991.

[Viehstaedt and Ambler 1992] G. Viehstaedt and A. Ambler, Visual representation and manipulation of matrices, *Journal of Visual Languages and Computing* 3(3), 273-298, Sept. 1992.

[Wadge and Ashcroft 1985] W. Wadge and E. Ashcroft, *Lucid, the Dataflow Programming Language*, Academic Press, London, 1985.

[Wang and Ambler 1996] G. Wang and A. Ambler, Solving display-based problems, *IEEE Symposium on Visual Languages*, Boulder, Colorado, 122-129, Sept. 1996.

[Wilde and Lewis 1990] N. Wilde and C. Lewis, Spreadsheet-based interactive graphics: from prototype to tool, *ACM Conference on Human Factors in Computing Systems*, 153-159, Apr. 1990.

[Wolber 1997] D. Wolber, Pavlov: An interface builder for designing animated interfaces, *ACM Transactions on Computer-Human Interaction* 4(4), 347-386, Dec. 1997.