DRAWINGS ON NAPKINS, VIDEO GAME ANIMATION, AND OTHER WAYS TO PROGRAM COMPUTERS

KEN KAHN (KAHN@CSLI.STANFORD.EDU)

Animated Programs
1748 Monticello Road
San Mateo, CA 94402

MAY 2, 1996 VERSION

This appeared the CACM special issue on new paradigms of computing, August 1996.

ABSTRACT

Programmers usually type characters on a keyboard to enter, test, and debug computer programs. More than 30 years ago researchers began augmenting those characters with diagrams [Sut66] but pure ASCII text is still the ubiquitous standard. We will argue that these attempts to make programming more visual have failed to become mainstream because they are too conservative.

Various radical syntaxes for programs are feasible and offer many advantages over the state of the art. Programs, for example, can be defined by the topology of sketches, even hand-drawn scanned sketches [Kah90]. A programming environment can parse these sketches and generate animations of the drawings evolving as the program executes [Kah92]. Or programs can be defined by manipulating physical objects, e.g., by connecting blocks together [Suk95] or inserting plastic cards into slots [Per76]. We are currently building a system called ToonTalk in which programs are created, run and debugged in a manner that closely resembles playing a video game [Kah96]. In the near future we may see program development systems that exist only in virtual reality or ones that interpret gestures in the real world.

Program sources need not be static collections of text or even text and pictures, but can be animated, tactile, enhanced with sound effects, and physical.

RADICAL SYNTAXES FOR PROGRAMMING LANGUAGES.

This paper is about syntax—a topic usually associated with heated battles about whether a comma or semi-colon is more desirable to separate certain program fragments. The syntax of programs, i.e., the *form*, not the semantics, has an incredible range that has barely been explored. For the purposes of exploring a wide range of syntaxes, we consider syntax to include both notation and syntactic structures. In addition to the traditional use of text to express programs, one can use pictures, animations, physical objects, gestures, and actions. Programs can be perceived by more than our eyes—our ears and sense of touch can play important roles.

The primary role of syntax in programming languages is to aid in the *communication* of a program from a human to a computer. The designers of syntaxes are concerned with issues of ambiguity, conciseness, ease of learning and use, ease of parsing, and readability. Readability has grown in importance due to the realization that people spend lots of time reading their programs and those of others. Literate programming [Knu92] addresses the readability issue directly by providing rich text formatting to program sources.

Before computer terminals became available, program sources were created on paper (often paper specially designed for this purpose—coding sheets) and then copied to punch cards or paper tape. The programs were compiled and executed and a paper listing was returned to the programmer. This kind of communication is limited to few exchanges per day. When computer terminals became widespread, much more fine-grained communications became possible. Interpreters and incremental compilers followed, as did structure or syntax-aware program editors. Programming became more conversational and less like exchanging letters. Strings of characters, however, remained as the foundation of programming language syntax.

Researchers have experimented for over thirty years with language syntaxes that add pictures, icons, and diagrams to these strings of characters to produce visual languages. [Mye90] These languages resemble flow charts, dataflow diagrams, and graphical re-write rules. This is an active field of research which has produced many innovative systems. It is fair to ask, however, why none of these languages or ideas have succeeded in any large-scale manner. (Note that Visual Basic and similar languages have a textual syntax and are visual only in the manner in which one programs windows, menus and buttons.) Perhaps visual programming needs decades of research before it is mature enough for wide-spread use. Virtual reality and hypertext took that long to make it to the mainstream. Maybe textual languages have a huge advantage because they were first. Textual languages enjoy a huge infrastructure in editors, lint programs, search tools, formatters, and integrated development environments. Some argue that if only a large engineering effort were placed behind a good visual programming language it would succeed.

But the problems with visual programming may be deeper. Some argue that visual languages are wasteful of precious screen real estate. And the languages don't scale well. That programming is an abstract symbolic activity and as such fits better with text than pictures. Others argue that syntax is relatively unimportant and that the small advantages of visual programming are not enough to move the installed base of programmers from what they currently know. Some see the value in using diagrams to illustrate aspects of computation (e.g. Petri nets, Turing machines, finite-state diagrams) but believe that diagrams are poorly suited for precisely capturing all the details that are necessary to encode programs.

Much of the debate about whether visual programming is better than text-only programming and the debates about which visual programming language is better are really debates about human psychology. And the participants are not experts. Only a few psychologists have studied how people cope with different alternatives in textual programming languages, and more recently have done comparative studies of textual and visual programming. [GP92, GP96] The more general issue of the role of visual thinking in math, logic, and science has been studied by psychologists for more than a century.

The position taken in this paper is that visual programming has failed to become wide-spread because it isn't *radical* enough. Programs describe *dynamic* processes. Visual programming languages attempt to *encode* descriptions of these dynamic processes with *static* pictures. Dynamic pictures, or animations, are a much better fit. Also, the process of encoding introduces abstraction. The cognitive advantages of the concreteness of pictures is mitigated by the fact that these pictures are components of a formal symbolic system. In pop psychology terminology, visual programming is well-suited for those people that are visual thinkers *and* adept with abstractions. Unfortunately, such people are apparently a small minority of the population.

There are visual languages that avoid the complexities of abstraction at the price of power and expressiveness. *KidSim* [Smi94], for example, lets programmers describe computations as graphical re-write rules. While a wide range of programs can be expressed this way, to achieve general-purpose programming KidSim's textual "properties" sub-language must be used. *Klik & Play* is a game maker published by Maxis that is promoted as game programming without having to program. Built-in objects have properties sheets from which users can select and customize a large range of behaviors and reactions to events. A surprising variety of programs can be constructed this way, but it is far from general-purpose computing. VisiCalc and subsequent spreadsheet programs can be thought of as special-purpose programming with a syntax based upon a grid layout. [Kay84] *Rocky's Boots* and *Robot Odyssey* were two games from The Learning Company in the early 1980s that supported an animated video-game syntax for building and simulating arbitrary logic circuits. There many other examples of innovative syntaxes in special-purpose programming systems. The focus of this paper, however are syntaxes of general-purpose programming languages.

The interplay between syntax and semantics in the design of programming languages is poorly understood. For example, a video-game animation syntax for a concurrent constraint programming language [Kah96] is presented below. It is hard to imagine a good video-game animation syntax for a traditional language like C or Pascal. Similarly, a dataflow diagram syntax fits a functional computational model and not a production system, while graphical re-write rules fit production systems and not functional computation. The interaction is equally complex at the level of individual language features. Lexical scoping, for example, can be easily captured by a picture syntax that supports containment and is difficult to capture well by a syntax based upon icons and lines between them.

PHYSICAL OBJECTS AS SOURCE CODE.

Programs can be more than text, more than pictures – they can be tactile. A program can be constructed by manipulating items in the real world so long as the computer is able to sense their state. In the mid-1970s, for example, Radia Perlman and Danny Hillis built a syntax device for a subset of the Logo programming language called a *slot machine*. [Per76] The prototype consisted of four units which differed only in their color. Each unit had ten or so slots into which plastic cards could be inserted. The cards had holes in the bottom so the machine could identify them. Some cards had pictures of basic operations (for Logo turtle programming these included forward, turn right, etc.). Others had numbers which could be placed in a slot together with an operation card to provide an argument to the operation (e.g., how far to turn right). Some just had a color that matched the color of one of the units. This provided for a means of expressing procedure calls. While the prototype had no way of expressing variables or conditionals, there were detailed plans for how to add them. The machine was limited to expressing numbers less than 10, procedures could not be longer than 10 instructions, and the planned variables were limited to numeric values. Despite these limitation it was capable of expressing an interesting class of programs.

As programs constructed on the slot machine ran, a light lit up on the slot currently executing. This is a simple but effective example of *program animation*. [Bro87] In program animation, as a program runs one sees representations of its state evolve. In most systems, the animation is not in the same visual terms as the program source but is instead a user-defined abstraction of the program state. On the slot machine, the lights

directly showed the state of the machine. Another interesting aspect of the machine was that it only read the cards when the slot was active. A programmer easily could remove and add cards as the program executed.

More recently a group of researchers at NEC have built what they call *AlgoBlocks*. [Suk95] These are cubes with electronics inside and connectors on the sides. Programs are constructed by connecting together blocks. A major focus on this project is the support collaborative programming where several people can sit at a table and jointly build a program. See Figure 1.

Figure 1 – Children constructing a program with AlgoBlocks.

One might wonder why one would want to program by manipulating physical objects. In both of these projects the answer is that they are well-suited for teaching programming concepts – even to very young children. Many children and some adults seem to learn better when there is something they can touch and manipulate. These research prototypes are rather limited but we can imagine professional programmers building programs by manipulating real world objects (other than the keyboard and mouse). There are clear disadvantages – one can't save or copy programs, editing is tedious, and it is hard to scale to large programs. Improved technology may help. More powerful systems might include cameras that observe what is being constructed or changed, objects than can move (or be moved by robots), and embedded electronics capable of giving the objects sophisticated behaviors. Perhaps manipulating physical objects will be the preferred method for constructing and testing small program fragments, and other means will be used for saving and combining them.

THE TOPOLOGY OF DRAWINGS AS SOURCE CODE.

Visual programs are nearly always constructed using a dedicated editor that is part of the language's program development environment. This severely limits the range of appearances for programs. An exception is Pictorial Janus [KS90b,Kah92] in which the syntax is defined in terms of the topological relations between picture elements. Relations like "inside", "touching" and "connected" are used in defining the syntax of the language. (See Figure 2.) Shape, color, size, and texture are left for programmers to use as they see fit.

Programs can be drawn on paper, scanned in, and parsed. (See Figure 3.) Or constructed using one's favorite illustration program. More radical possibilities involve constructing program fragments with plastic Color Forms® toys, Lego® bricks or other physical objects and then capturing their appearance by a scanner or camera. A friendly programming environment could constantly interpret the input from cameras and give advice and help.

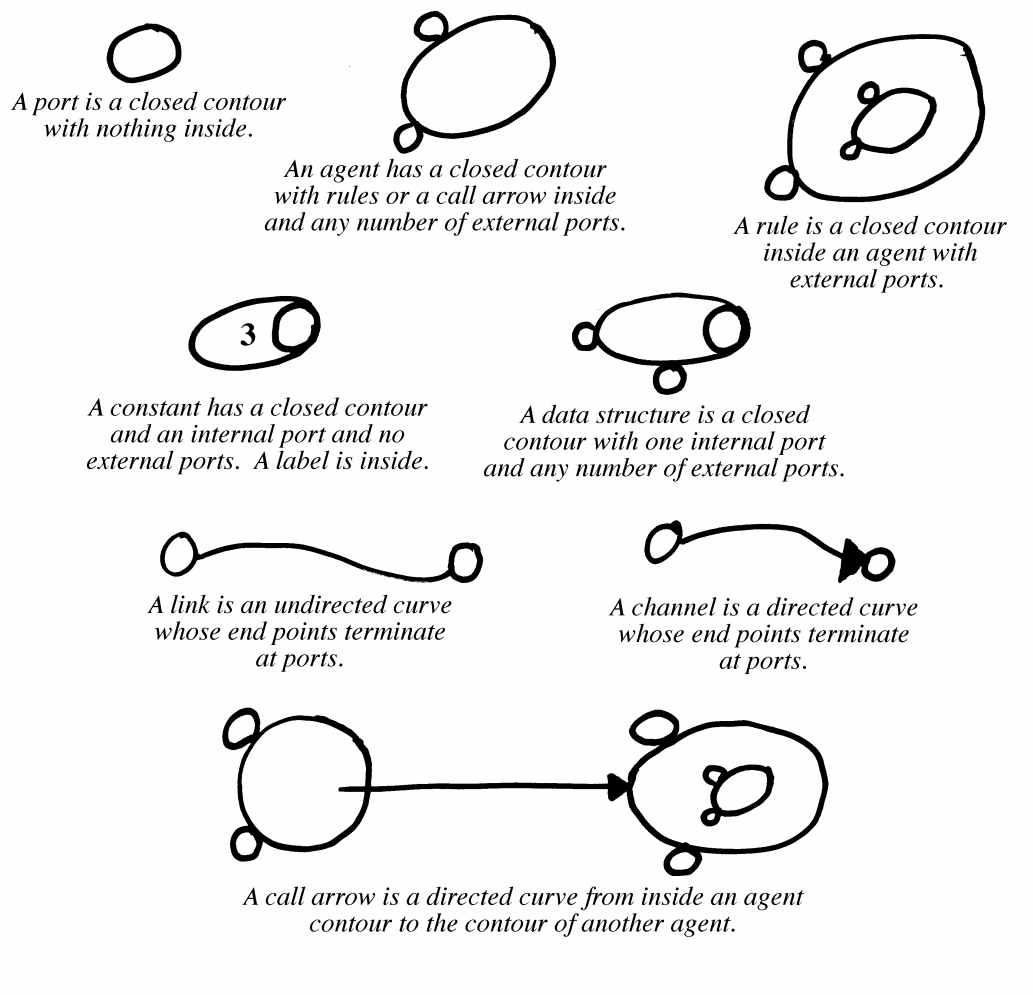


Figure 2 – The topological syntax of Pictorial Janus

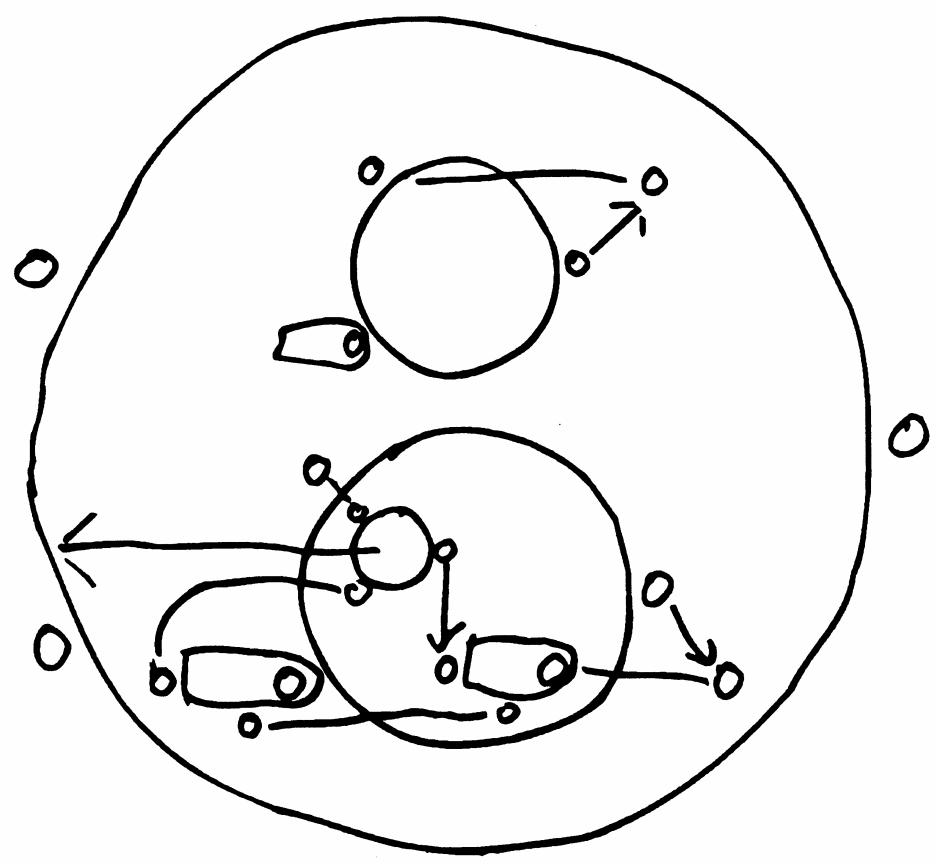


Figure 3 – Hand-drawn source code for appending lists in Pictorial Janus.

Pictorial Janus also animates the execution of programs in the same visual terms they were constructed. A hand-drawn program, unless automatically cleaned up, will animate as hand-drawn elements that grow, shrink, move, and dissolve. The animation of an agent reduction shows a rule expanding until it visually matches the agent contour. It then dissolves away leaving behind the body of the rule. Links shrink as newly created agents grow. (See Figure 4.)

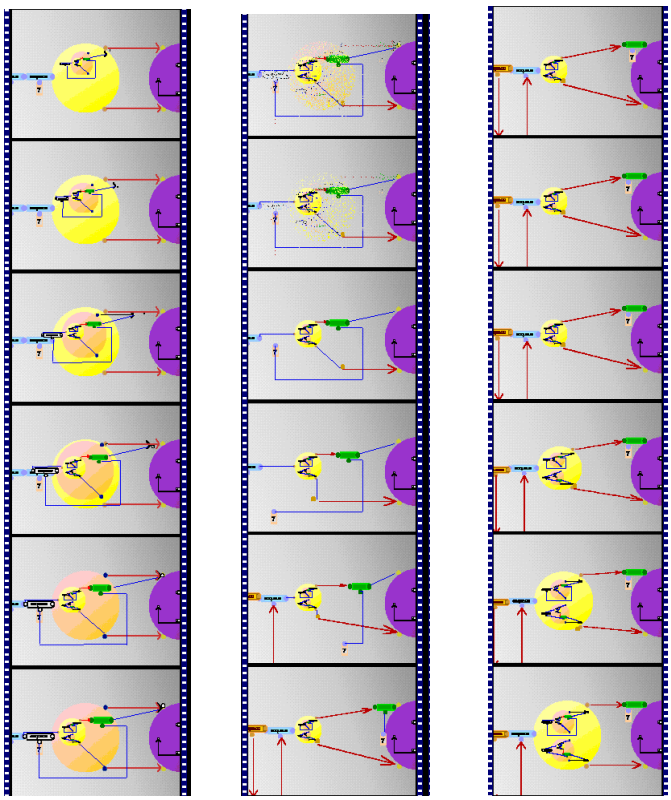


Figure 4 – Snapshots of a single reduction of a Pictorial Janus queue program.

Reliance upon topological features of drawings simplifies greatly the parsing problem. It gives the programmer a great deal of freedom – freedom that, if abused, can lead to hard to understand diagrams. Unlike simple picture parsers based upon topology, the human visual system has difficulty abstracting. Two items with similar shapes and colors are easily confused even if they differ in some significant topological aspect. Programmers need to use the graphical freedom with care – just as users of word processors need to use care when mixing lots of different fonts in the same document. In both cases, good styles emerge and should be followed.

A more fundamental problem that Pictorial Janus shares with most visual programming systems is that formal diagrams are used to encode programs and many people find formal diagrams difficult to understand and construct. Venn diagrams, for example, are much simpler than visual programs and while most readers of a journal like *Communications of the ACM* find them very easy, one forgets how hard it is for most children to learn them.

VIDEO-GAME ANIMATION AS SOURCE CODE.

Animation is very well-suited for displaying computations, not just because it is dynamic, but because it can elide unimportant aspects and highlight other aspects. The human visual system is well-tuned for perceiving events and patterns in changing images. Since animation is well-suited for showing a computation as it evolves, could it also be well-suited for program sources?

ToonTalk™ is a programming "language" whose source code is animated. [Kah96] (ToonTalk is so named because one is "talking" in (car)toons.) This does not mean that it is a visual programming language where some static icons have been replaced by animated icons. It means that animation is the means of communicating to both humans and computers the entire meaning of a program. While the advantages of animated source code are many, constructing animation is generally difficult and time-consuming. Good animation authoring tools help but it is still much more difficult to animate an action than to describe it symbolically.

Luckily, there is one sort of computer animation that is trivial for a user to produce—video game animation. Even small children have no troubles producing a range of sophisticated animations when playing games like Nintendo's *Mario Brothers*®. While the range is, of course, very limited relative to a general animation authoring tool, video game style animation is fine for the purposes of communicating programs to computers. If, for example, a program fragment needs to swap the values of two locations, what can be more natural and easy than grasping the contents of one, setting it down, grasping the contents of the other, placing it at the first location and then moving the original item to the second location? (See Figure 6.) This is something a very young child can understand and do, while only a programmer can write the equivalent code. (See Figure 5.)

```
temp = x;
x = y;
y = temp;
```

Figure 5 – Swapping two items in C

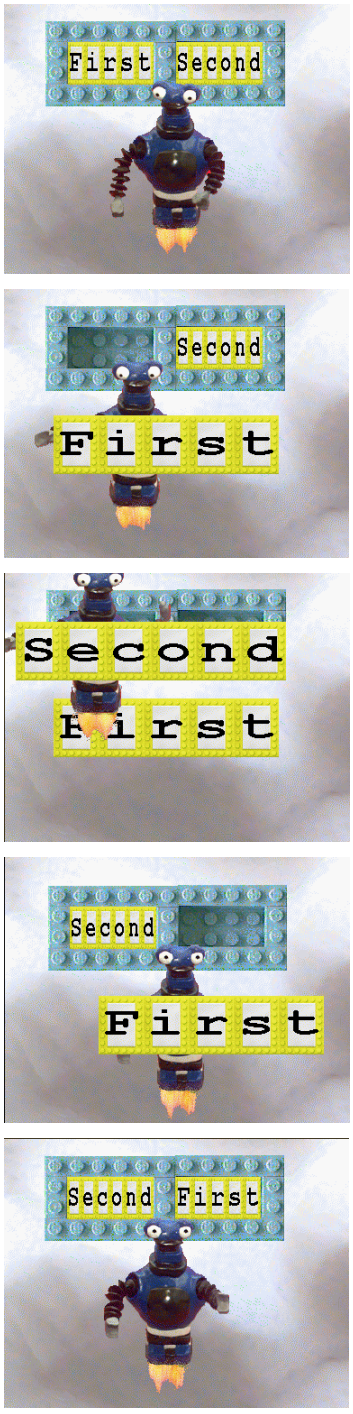


Figure 6
Snapshots of the creation of a ToonTalk program to swap two items.

Once the step is taken to use video game technology for the construction of source code, it is easy to see other uses of video game technology for browsing, editing, executing and debugging programs. Other ideas from video games can be borrowed. Some video games have animated characters whose purpose is to provide help to users. These characters can play the role of on-line help and tutorial systems.

Computer scientists strive to find good abstractions for computation. In ToonTalk a critical goal is to find good "concretizations" of those abstractions. The challenges are twofold: to provide high-level powerful constructs for expressing programs and to provide concrete, intuitive, easy-to-learn, systematic game analogs to every construct provided. After all, a Turing Machine is both concrete and a universal computing formalism. (Alan Turing strove not just for mathematical computing formalisms but for their concrete analogs as well.) One can imagine a Turing Machine game that in theory supports the construction of arbitrary computations, but I can't imagine using it to build large or complex programs.

The ToonTalk world resembles a twentieth century city. There are helicopters, trucks, houses, streets, bike pumps, toolboxes, hand-held vacuums, boxes, and robots. Wildlife is limited to birds and their nests. This is just one of many consistent themes that could underlie a programming system like ToonTalk. A space theme with shuttle craft, teleporters, and the like would work as well. So would a medieval magical theme or an Alice in Wonderland theme.

An entire ToonTalk computation is a city. Most of the action in ToonTalk takes places in houses. Communication among houses is accomplished by homing pigeon-like birds. Birds accept things, fly to their nest, leave them there, and fly back. Typically houses contain robots that have been trained to accomplish some small task. A robot is trained by entering into its "thought bubble" and showing it what to do. The robot remembers the actions in a manner that can easily be abstracted to apply in other contexts.

Robots working in different houses is the source of concurrency in ToonTalk. Synchronization is accomplished in ToonTalk by giving robots boxes where there are nests where they expect some other item. Such robots will wait until a bird flies in and covers a nest by an item before trying to proceed. If a robot expects to be given a particular kind of box (e.g., with one hole which contains the number zero) and another item is in the box (e.g., the number one) then it will give the box to the robot behind it in line.

The behavior of a robot is exactly what it was trained to do by the programmer. This training corresponds in traditional terms to defining the body of a method or clause.

- sending a message by giving a box or pad to a bird,
- spawning a new agent by dropping a box and a team of robots into a truck (which drives off to build a new house),
- performing simple primitive operations such as addition or multiplication by building a stack of numbers (which are combined by a small mouse with a big hammer),
- copying an item by using a magician's wand,
- terminating an agent by setting off a bomb,
- changing a tuple by taking items out of compartments of a box and dropping in new ones.

Figure 7 – Possible actions of ToonTalk robots.

These correspond to the permissible actions of a concurrent logic programming agent or an actor [KS90a, Agh87]. The last one may appear to introduce mutable data structures into the language, which are known to introduce much complexity to parallel programs. Since boxes, however, are copied, not shared, this is not the

case. An apparently destructive operation on a private copy is semantically equivalent to constructing the resulting state from scratch. But the destructive operation is often more convenient. In situations where copying is inappropriate, a house can be built to hold a single copy and multiple references can be accomplished by copying birds which deliver requests to a shared nest in that house.

When the user controls the robot to perform these actions, she is acting upon concrete values. This has much in common with keyboard macro programming and programming by example [Smi75]. (Read Lieberman's companion paper in this issue for a more thorough discussion of learning from examples. [Lie96]) The hard problem for programming by example systems is how to abstract the example to introduce variables for generality. ToonTalk does no induction or learning. Instead the user explicitly abstracts a program fragment by removing detail from the thought bubble. The preconditions are thus relaxed. The actions in the body are general since they have been recorded with respect to which compartment of the box was acted upon, not what items happened to occupy the box.

One can understand ToonTalk completely in its own terms. E.g., a bird, when given something, flies to her nest, leaves the item there and returns. This is how children typically understand it. Computer scientists, however, might like to understand the relationship between computation and these ToonTalk objects and activities. Here's the mapping between computational abstractions and ToonTalk's computational concretizations:

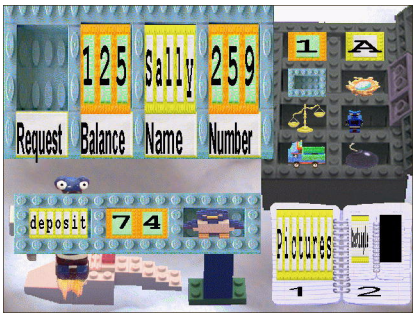
Computational	**ToonTalk**
computation	city
agent (or actor or process or object)	house
methods (or clauses or program fragments)	robots (with thought bubbles)
method preconditions	contents of thought bubble
method actions	actions taught to robot inside thought bubble
tuples (or arrays or vectors or messages)	boxes
comparison tests	scales
agent spawning	loaded trucks
agent termination	bombs
constants	number pads, text pads, pictures
channel transmit capabilities	birds
channel receive capabilities	nest
program storage	notebooks

A problem with an animated syntax is the difficulty of presenting programs on paper. For example, Figure 9, shows snapshots from the construction of a program fragment that implements a simple concurrently accessible bank account. In the example, a robot is trained to accept a request to deposit and sends back an acknowledgment after the request is processed. The robot adds the amount of the deposit to the current balance, acknowledges the request by responding with a copy of the new balance, and then becomes ready to process the next request. ToonTalk is capable of describing what a robot does in full detail, but as can be seen in Figure 8 the text is difficult to follow. The difficulty is only partially due to the simplicity of ToonTalk's text generator. Narratives of sequences of these kinds of actions are just hard to follow. In contrast, an animation of the same sequence is easy for even a small child to understand.

> This is a robot who wants a box with 4 holes whose hole labeled with "Request" contains a box with 3 holes whose first hole contains a pad with the text "deposit" on it and whose hole labeled with "Amount" contains any number and whose hole labeled with "Ack" contains a bird and whose hole labeled with "Balance" contains any number and whose hole labeled with "Owner" contains a text pad with anything on it and whose hole labeled with "Number" contains any number. If given a box like that he will pick up what's in the first hole of the box he's working on. And drop it. And pick up what's in the second hole of the first thing he made or found. And drop it on the second hole of the box he's working on. And grab a magic wand. And use the magic wand on the second hole of the box he's working on. And give what he just copied to the bird in the third hole of the first thing he made or found. And release the magic wand. And grab a copy of Dusty. And use Dusty to vacuum the first thing he made or found. And release Dusty.

Figure 8 – Description generated by ToonTalk of the robot handling bank deposit requests

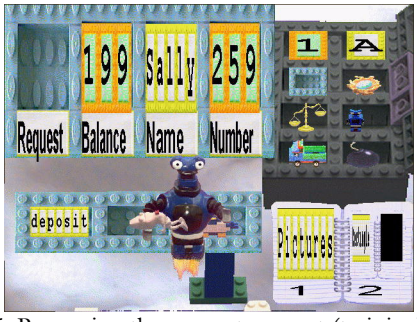
1. Giving robot box to work on



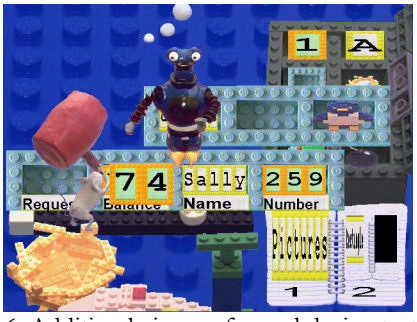
2. Taking out the current request (training)

3. Adding deposit to balance (training)

4. Giving copy of balance to bird (training)



5. Removing the current request (training)



6. Addition being performed during test

Figure 9 – Snapshots of a ToonTalk robot being trained to handle bank account deposit requests

WHO IS THE "CUSTOMER" OF PROGRAMMING LANGUAGE SYNTAX?

A popular corporate fad is to be "customer-driven" and to constantly ask of every activity "Who is the customer?". In this spirit we ask who is the customer of syntax. It is generally believed that a professional programmer requires a very different syntax than someone just learning to program. For example, the Logo programming language for children borrowed heavily from Lisp but completely redesigned the syntax to be easier to learn and read. But there are noteworthy examples of syntaxes used by beginners and professional alike. Basic, for example, was designed for novices and is now used by millions of professionals (who use Visual Basic, LotusScript, Word Basic, etc.). And many people learn C as their first programming language despite its complex syntax.

Can the radical syntaxes presented in this paper also have a broad range of "customers" or are they primarily for children and beginners? To answer this question one needs to look case by case and consider whether design compromises were made between ease of use and ease of learning. Pictorial Janus, for example, is relatively difficult to learn but is very flexible. The syntax of ToonTalk is so easy to learn that 6 years olds master it quickly. Not only is it easy to learn but it is perhaps unique in that it is *fun* to learn and use. Some children play with ToonTalk but construct nonsense programs because it is fun to train robots, use magic wands, give birds things and so on. Many think that anything that is fun cannot also be an appropriate tool for professionals. It is hard to imagine system programmers in the future programming by loading up trucks, using hand-held vacuums and bike pumps, and the like. If this is the only reason an animated video-game syntax is inappropriate for professional use, one can make a bland variant of ToonTalk.

Another "customer" of a syntax is a person with mental or physical disabilities. For example, a programmer unable to control his or her body could construct ToonTalk programs with an eye tracker or head tracker for input. Or a dyslexic programmer could draw programs or construct them in a video game world in which reading is not essential. ToonTalk was designed to be usable with just a game controller – a keyboard is unnecessary. It is this that makes it suitable for users with very limited ability to move.

Another consideration is how long it takes to construct programs. For example, a good typist can enter a program to append two lists together in a functional or logic programming language in about 60 seconds. Someone familiar with Pictorial Janus can draw on paper the equivalent program in the same amount of time. With an illustration program it can take 2 or 3 times longer. With a Pictorial Janus editor the time can be significantly less. In ToonTalk the equivalent program can be constructed in about 3 minutes. However, in ToonTalk the process of constructing a program includes constructing test data and testing program fragments as they are built. All these times are for someone who already knows very well how to append two lists. Anecdotal evidence suggests that when one is less sure what one is doing that ToonTalk can be faster than other alternatives since most people can reason more easily with concrete sample data rather than abstract variables. Also, one sees immediately the result of actions and many bugs are caught immediately after they are created.

Yet another consideration is how easy it is to manage and modify large programs. Here text currently has a large advantage. Nearly all text-based program editors support some kind of search and replace. Tools like "grep" are available for searches across many files. Similar tools for illustrations and diagrams are more difficult to build and have not passed beyond the research prototype phase [Kur93]. ToonTalk currently has *no* editor at all. If a robot was trained incorrectly, then a new robot must be trained to replace the bad robot. A syntax that relies exclusively upon physical objects results in programs that are *very* hard to search. Editing on a slot machine is so simple that four year old children mastered it easily [Per76], but it is slow and tedious to make certain changes like inserting an instruction. Hybrid syntaxes and putting electronics and behaviors into the objects can alleviate many of these shortcomings. The important question, however, is not what tools currently exist but whether there are fundamental obstacles to building tools that are competitive with text-

based tools. One can imagine, for example, a ToonTalk robot editor that shows the actions the robot takes as a "story board". Story boards could be edited with cut, copy, and paste. How far such tools can come is an open question that requires much more research.

A related question is how effectively a particular syntax uses screen real estate. This is a common criticism of visual programming systems. Due to Pictorial Janus' reliance upon topology, the size of a program fragment is not constrained. In particular, the zoom feature of most illustration programs means that programs can be smaller than a pixel at the standard zoom factor. If one compares equivalent textual and Pictorial Janus programs constrained to be easy to read then they use about the same area on the screen. In ToonTalk programs exist in time, not space, so the question is not applicable. Physical objects that are easy for humans to manipulate need to be relatively big. "Desk real estate" may be a real constraint. It is interesting to note, however, that AlgoBlocks could have been designed to be much smaller, but the designers were concerned that that would interfere with collaborative programming.

A programming language syntax is usually thought of as a description of the form of a program and not of the state of a computation. The slot machine, Pictorial Janus and ToonTalk all provide a means of seeing the state of an entire computation in the same visual terms or syntax as the programs being executed. The state can be seen in the slot machine lights, the Pictorial Janus configurations of agents, and the ToonTalk city alive with trucks driving, birds flying, and houses being constructed and destroyed. The latter two show the state of a computation as a large space which can be navigated.

Since syntax is used for communication of programs to computers, another "customer" is the computer itself. From this viewpoint, a syntax should be precise, unambiguous, and quick to parse. Formalizing radical syntaxes is a challenge (e.g. [LM94] and [Haa95]). Except for Pictorial Janus, this isn't a practical problem since in other visual programming systems and in ToonTalk, programs can only be constructed using a specialized piece of software. This has the unfortunate consequence that programs printed on paper can be ambiguous.

THE FUTURE OF PROGRAMMING LANGUAGE SYNTAX.

The purpose of this paper is to describe the incredible breadth of ways that programs can be expressed, ranging from sketches on paper, to video-game animation, to manipulation of physical objects. This space has barely been explored. Programs could be constructed from *inside* virtual reality, for example. Actions in virtual reality could be interpreted in much the same manner as actions in ToonTalk. Or a programmer could make gestures in front of a camera connected to a computer. Software could interpret these gestures as program instructions.

Audio can play a large role in action-oriented syntaxes and in program animation. ToonTalk, for example, has nearly 50 different sound effects that are associated with different actions and events. Speech output could be used to provide feedback that actions or gestures of the programmer are being interpreted correctly. Speech input could be used to accelerate the drawing of Pictorial Janus programs or the control of ToonTalk objects and tools. Maybe even music could be useful.

We foresee a future where programmers will use a combination of media and devices for constructing, running, testing, and debugging their programs. Text won't be obsolete but will be joined by sketches, animations, sound effects, speech, tactile feedback devices, gestures, virtual reality, cinemagraphic techniques, and electronic gadgets to help a wide range of people build, test, and debug computer programs.

ACKNOWLEDGMENTS.

I wish to thank Mary Dalrymple, Ted Selker, and Henry Lieberman for their insightful comments on earlier versions of the paper.

HOW TO OBTAIN SOFTWARE DISCUSSED.

Pictorial Janus software is available for non-commercial purposes from ftp://ftp.parc.xerox.com/pub/PictorialJanus and http://www.cadlab.de/~wolfgang/wm.visual.html.

ToonTalk is currently in beta testing. Visit http://www.toontalk.com or send email to KenKahn@ToonTalk.com if you wish to learn more about ToonTalk or to obtain a beta copy.

REFERENCES.

[Agh87] G. Agha, *Actors: A Model for Concurrent Computation in Distributed Systems*. The MIT Press, 1987.

[Bro87] M. Brown. *Algorithm Animation*, The MIT Press. 1987.

[GP92] T. R. G. Green and M. Petre. "When Visual Programs are Harder to Read than Textual Programs", *Sixth European Conference on Cognitive Ergonomics*, 1992.

[GP96] T. R. G. Green and M. Petre. "Usability Analysis of Visual Programming Environments: a 'cognitive dimensions' framework", *Journal of Visual Languages and Computing*, Academic Press, Vol. 7, No. 2 , June 1996.

[Haa95] V. Haarslev. A Fully Formalized Theory for Describing Visual Notations*, Proceedings of the IEEE Visual Language Conference*, 1995.

[Kah90] K. Kahn and V. Saraswat. "Complete Visualizations of Concurrent Programs and their Executions", *Proceedings of the IEEE Workshop on Visual Languages*, October 1990, Skokie, IL.

[Kah92] K. Kahn. "Concurrent Constraint Programs to Parse and Animate Pictures of Concurrent Constraint Programs*", Proceedings of the Fifth Generation Computer Systems Conference*, Tokyo, Japan, June 1992

[Kah96] K. Kahn. "ToonTalk™ -- An Animated Programming Environment for Children", *Journal of Visual Languages and Computing*, Academic Press, Vol. 7, No. 2 , June 1996.

[Kay84] A. Kay, "Computer Software", *Scientific American*, Vol. 251, No. 3, pp. 41-47, September 1984.

[Knu92] D. Knuth. *Literate Programming*. Center for the Study of Language and Information, Stanford University. 1992.

[KS90a] Kenneth Kahn and Vijay Saraswat. Actors as a special case of concurrent constraint programming. In *Proceedings of the Joint Conference on Object-Oriented Programming: Systems, Languages, and Applications and the European Conference on Object-Oriented Programming*. ACM Press, October 1990.

[KS90b] Kenneth M. Kahn and Vijay A. Saraswat. Complete visualizations of concurrent programs and their executions. In *Proceedings of the IEEE Visual Language Workshop*, October 1990.

[Kur 93] D. Kurlander. *Graphical Editing by Example*. Doctoral Dissertation,

Computer Science Department, Columbia University. July 1993.

[LM94] G. Lehrenfeld and W. Müller. "Defining the relational grammar of PJ - a case study", Technical Report cr-07-94, Cadlab, Paderborn University, Germany, 1994.

[Lie96] H. Lieberman. "Intelligent Graphics: A New Paradigm at the Intersection of AI and Computer Graphics", *this issue of CACM*.

[Mye90] B. Myers. "Taxonomies of Visual Programming and Program Visualizations", *Journal of Visual Languages and Computing*, Academic Press, Vol. 1, pp. 97-123. 1990.

[Per76] R. Perlman. "Using computer technology to provide a creative learning environment for preschool children". MIT AI Lab Memo 360. Logo Memo 24. May 1976.

[Smi75] David Smith, *Pygmalion: A Creative Programming Environment*, Stanford University Computer Science Technical Report No. STAN-CS-75-499, June 1975. Also Stanford University Ph.D. thesis.

[Smi94] David Smith, Allen Cypher and Jim Spohrer, KidSim: Programming Agents without a Programming Language, *Communications of the ACM*, Vol. 37, No. 7, July 1994.

[Sut66] W. Sutherland. *On-line Graphical Specification of Computer Procedures*, Ph.D. thesis, MIT, 1966

[Suk95] H. Suzuki and H. Kato. Interaction-Level Support for Collaborative Learning: AlgoBlock – An Open Programming Language. *CSCL95 Proceedings*, October 1995.