Seeing Systolic Computations in a Video Game World

Ken Kahn
A paper submitted to VL'96: IEEE Symposium on Visual Languages, February 12, 1996
(Published September 1996 version not available)
Animated Programs
San Mateo, California

Abstract

ToonTalk™ is a general-purpose concurrent programming system in which the source code is animated and the programming environment is like a video game. Every abstract computational aspect is mapped into a concrete metaphor. For example, a computation is a city, an active object or agent is a house, inter-process communication represented by birds carrying messages between houses, a method or clause is a robot trained by the user and so on. The programmer controls a "programmer persona" in this video world to construct, run, observe, debug, and modify programs.

ToonTalk has been described in detail elsewhere [1]. Here we show how systolic programs can be constructed and animated in ToonTalk. Systolic computations run on multiple processors connected in a regular topology, where all communication is via local message passing [2]. A ToonTalk city can be seen as a systolic multi-processor and each house in the city as an active processor. One is able to construct systolic algorithms and watch their execution as houses are built and destroyed (i.e., processes are spawned and terminate) and birds carry messages between houses.

A brief introduction to ToonTalk

Video games, especially adventure and role-playing ones, place the user in an artificial universe. The laws of such universes are designed to meet constraints of game play, learnability, and entertainment. While playing these games the user learns if doors need keys to open, if one's health can be restored by obtaining and consuming herbs, and so on. What if the laws of the game universe were designed to be capable of general purpose computing, in addition to meeting the constraints of good gaming?

It seems that no one has ever tried to do this. *Rocky's Boots* and *Robot Odyssey* were two games from The Learning Company in the early 1980s that excited many computer scientists. In these games, one can build arbitrary logic circuits and use them to program robots. This is all done in the context of a video game. The user persona in the game can explore a city with robot helpers. Frequently in order to proceed the user must build (in an interactive animated fashion) a logic circuit for the robots to solve the current problem. ToonTalk is pushing the ideas behind *Robot Odyssey* to an extreme, capable of supporting arbitrary user computations (not just the Boolean computations of *Robot Odyssey*).

There has been much research on *demonstrative* or programming by example systems. Perhaps the earliest was Pygmalion, built in the mid 1970s by David Canfield Smith as part of his doctoral thesis [11]. He even spoke of programs as "movies". But there was no animated game-like world in which the programmer operated. Instead, icons and menus were selected and a sequence of such selections could be iconized.

Computer scientists strive to find good abstractions for computation. In ToonTalk, in addition, we are striving 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 it's hard to imagine using it to build real programs.

The ToonTalk world resembles a twentieth century city. There are helicopters, trucks, houses, streets, bike pumps, toolboxes, hand-held vacuums, and robots. Wildlife appears in the form of 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 which when given things, fly to their nest, leave them there, and fly back. Typically houses contain robots that have been trained to accomplish some task. Robots have thought bubbles that contain pictures of what the local state should be like before they will perform their task. Local state is held in cubby holes (i.e., boxes). Cubbies also are used for messages and compound data (i.e., tuples). If a robot is given a cubby containing everything that is in its thought bubble, it will proceed and repeat the actions it was taught. Abstraction arises because the picture in the thought bubble can leave things out and it will still match. A robot corresponds roughly to a method in an object-oriented language or a conditional. A line of robots provides something like an "if then else" capability. Animated scales placed in a compartment of a box will tip down on the side whose neighboring compartment is greater than (or if text, alphabetically after) the compartment on the other side. By using scales tipped one way or another, the conditionals can include less than, equal to, or greater than tests.

Synchronization is accomplished in ToonTalk by the fact that if a robot is given a cubby and its thought bubble includes items that aren't yet in the cubby it will wait around until those items appear. An empty hole may be filled or a bird may cover a nest in a hole by an item. If the robot expects to be given a particular kind of cubby (e.g., a box holding the number zero) and another item is in the cubby (e.g., the number one) then it will give the cubby 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. The possible actions are:

- 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 hammered together 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.

These correspond to the permissible actions of a concurrent logic programming agent or an actor [12, 13]. The last one may appear to a computer scientist 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 there would be a performance penalty from unnecessary copying of boxes, a house can be built to hold a single copy and sharing can be accomplished by copying birds which deliver requests to a nest in that house.

When users control the robot to perform these actions, they are acting upon concrete values. This has much in common with keyboard macro programming and programming by example [11]. 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.

If users never turned off their computers nor wanted to share a program with other users then this would be adequate. Notebooks provide permanent storage for anything they have built. Notebooks can contain notebooks to provide a hierarchical storage system. Notebooks provide an interface to the essential functionality of the file system without disrupting the ToonTalk metaphor. The initial notebook contains sample programs and access to facilities like animation and sounds. In addition to notebooks there is clipboard object to facilitate transfer of objects into and out of ToonTalk using the window system's clipboard feature.

Kids can understand ToonTalk completely in its own terms. E.g., a bird, when given something, flies to its nest, leaves the item there and returns. 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 object)	house
methods (or clauses)	robots
method preconditions	thought bubble
method actions	robot's actions
tuples (or arrays or messages)	boxes
comparison tests	scales
agent spawning	loaded trucks
agent termination	bombs
constants	number, text, picture
channel transmit capabilities	birds
channel receive capabilities	nest
program storage	notebooks

Systolic programming in Concurrent Prolog

Systolic programming is an approach for creating massively parallel processing. Originally intended for direct implementation in VLSI [3], it was generalized and recast as a software-oriented methodology of algorithm design and implementation by Shapiro and his colleagues at the Weizmann Institute of Science [2]. They took Concurrent Prolog, a concurrent variant of Prolog, and added a processor mapping notation.

They chose a notation based upon turtle geometry [4], a notation best known for its use in the Logo programming language for children [5]. (As acknowledged in [2], the use of turtle geometry was at the suggestion of the author of this paper.) The essence of the idea is that when specifying a process creation one could optionally add a fixed instruction turtle program as an annotation. The turtle program has no impact on the semantics of the underlying program – only upon the mapping of computational processes to processing nodes.

Computation occurrs upon a torus topology, i.e., a two-dimensional grid of processors each of whose edges are connected to the opposite edge. The turtle programs specify where a process should be spawned relative to the location (and heading) of the process that created it. For example, p@(forward(1),right(90),forward(1)) creates a process "p" at the processor that can be found by going forward one processor, turning right 90 degrees and going forward one processor again. The process "p" has a heading that is turned 90 degrees clockwise from the heading of the spawning process.

Various algorithms were implemented in this framework ranging from the Sieve of Eratosthenes for prime number generation, to matrix multiplication, to sorting, to Gaussian elimination, to solving the Towers of Hanoi puzzle. The corresponding process structures ranged from linear pipes to process arrays to process trees.

If a process structure is a good match for an algorithm, then communication between processes will require message passing only between neighboring processors. Furthermore, the load of processing required of the processing nodes will be well balanced.

The primary purpose of these process-to-processor mappings was to give programmers a notation for optimizing the execution of their programs on parallel processors. However, it was also noted that this notation enabled pedagogically effective *visualizations* of current algorithms. One could watch the output of a simulator that displayed processes being spawned and terminating and communication between processes. Process mappings enabled the viewer to see the large-scale structure of a computation as it evolved. An automatic load balancing system without a mapping notation is unlikely to produce recognizable patterns of compuation.

Systolic programming in ToonTalk

In ToonTalk, process spawning is accomplished by loading up a truck. Marty, the ToonTalk guide explains that there is a construction crew in the truck that is eager to build a new house for a robot (or a team of robots) to work in. When robots are placed in a truck together with a box of things for them to work on, they drive off and build a house as close by as possible. The crew places the robots on the floor of the new house and gives the first robot the box. A typical nine year old child within the first hour of using ToonTalk has no trouble building houses, and even training robots to build houses.

Recently ToonTalk was extended to give programmers more control over the construction of houses. Addresses were introduced: streets (which run east and west) and avenues (which run north and south). One can perform arithmetic operations on addresses to compute new addresses. Addresses can be given to a truck crew and they will build as close to that address as they can. It is easy to specify, for example, that the house should be built 3 blocks north and 2 blocks east of the current address. The addressing can easily be in terms of either absolute or relative Cartesian coordinates. Turtle geometry addressing is also possible but not easy.

Furthermore, ToonTalk was extended so that one can also tell the truck how the house should look. Currently this is restricted to the three different house styles natively supported by ToonTalk. This is useful for distinguishing different types of processes running simultaneously.

Computing Factorial – a simple example

Consider as an example the task of computing the factorial of an integer. This very simple example will be used to illustrate how one can create and animate systolic computations like those described in [2]. The most straight-forward way to compute factorial in ToonTalk is

iteratively. A team of robots in a single house can compute the result. To see the animation of such a computation one need only go into the house and watch the robots at work.

[Note to reviewers: the figures are perhaps difficult to read.. They will be improved in the final version.]

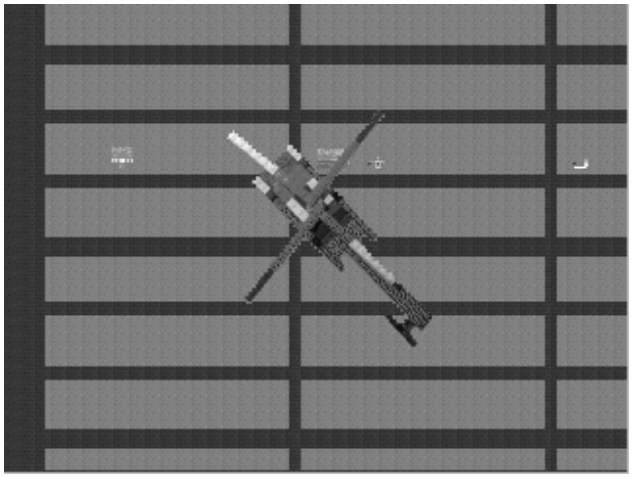

Figure 1
Screen shot of a town executing singly recursive factorial
(Houses are easier to see in color original.)

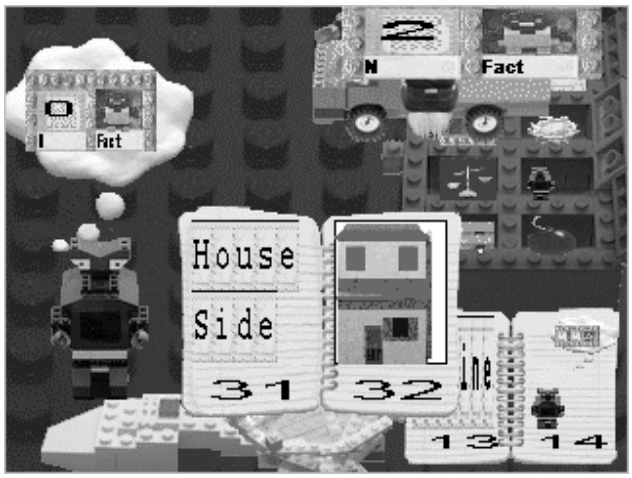

Figure 2
Screen shot of singly recursive factorial inside a house

Recursive factorial is a bit trickier. The robot that does most of the work needs to multiply "n" by the result of computing factorial of "n-1". We trained this robot to create a new house for the computation of factorial of "n-1" directly east of the current location. The robot that will wait for the result and multiply it by "n" is created at the current location but is given a different house style from the houses where factorial is being computed. As can be seen in Figures 1 and 2, as the computation proceeds a line of houses "n" long is built from left to right and then houses are destroyed from right to left. This mimics very closely the behavior of a stack implementation of recursive factorial.

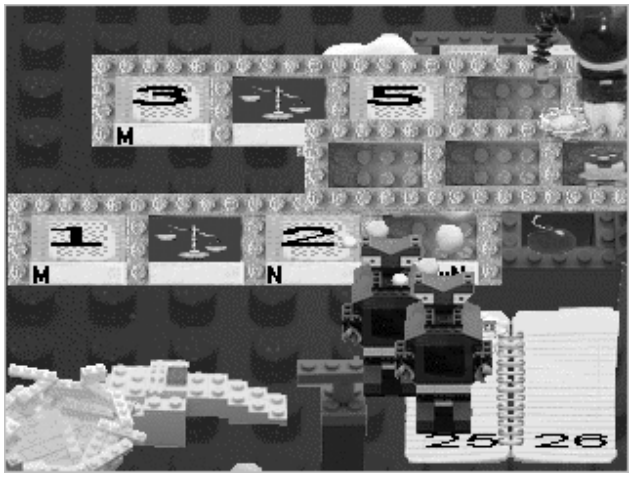

Figure 3 -- Screen shot of a robot in the midst of executing doubly recursive factorial.

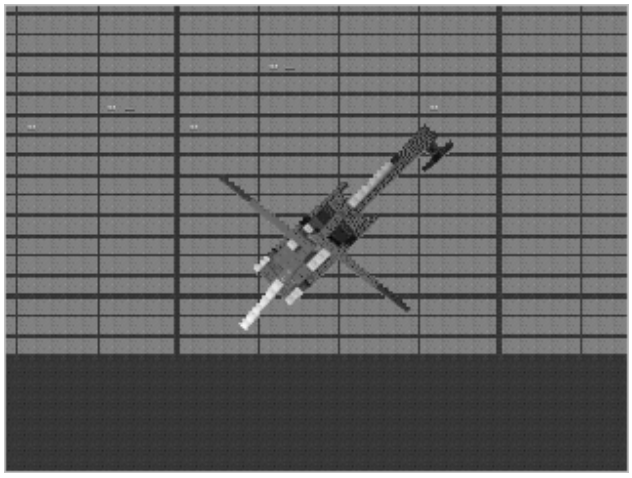

Figure 4 – Houses are being built in a tree structure by a robot computing doubly recursive factorial.

A more interesting example is a doubly-recursive factorial. A robot to compute factorial of "n" constructs a house to compute the product of the integers from 1 to "n". If "n" is greater than 1, then the main working robot in that house splits the problem to computing the product of 1 to "n/2" and to computing the product of "n/2+1 to n" and as before in another house a robot will wait for the results from these two houses and pass along the product. Figure 3 is a screen shot from inside a house as a robot is in the middle of breaking the product in two. As in [2] there are many choices for how to lay out the underlying tree of processes spawned when running this program. In Figure 4, we see a screen shot while flying over a city populated with robots who spawn the two sub-product computations, one to the left and the other to the right and both below its current location. To avoid collisions of

sub-trees the distance is a function of how many numbers are being multiplied. It is easy to observe that the same number of houses are required to run the doubly recursive version as the singly recursive one, but since robots can work in parallel in different houses, it can be computed in O(log(n)).

Discussion

With a simple extension to ToonTalk, we can now see patterns in large scale computations. We believe this makes it easier to understand how parallel algorithms work and provides intuitions as to their computational complexity.

Unlike the work of Shapiro and colleagues, ToonTalk provides a way to see computations at both the macroscopic and microscopic levels. For example, at any time while the doubly recursive factorial is running, the user can stop all the robots, land the helicopter, and enter a house on the fringe of the tree of houses and restart the robots. The details of how the factorial problem is split in two can then be observed. (Figure 3.)

While these new extensions to ToonTalk have yet to be used with children, the underlying mechanism is a small, and easy to understand, extension to what construction crews can do. While perhaps a doubly recursive factorial is too complex for a typical 10 year old, it may be appropriate for high school or college level computer science students.

There is another motivation for this enhancement of construction crews. Children like to have control over the ToonTalk city. Some children train robots to build houses with robots inside that do nothing. Their goal isn't always computational – sometimes it is just playfulness. Having more control over the construction of houses makes it more fun.

Related work

While there is a large body of work on program animation [6] and visual programming, little of it deals with observing large scale computations. Most of the work has focussed on visualizing the evolution of data structures as algorithms execute. Few attempt to show the process or program state. Other than Pictorial Janus [7] we are not aware of any systems that are both program construction tools as well as visualization tools that span the range from showing every computation step to showing an entire computation.

VISTA [8], Communicating Thread Animation [9], and [10] are noteworthy examples of systems for portraying very large-scale current computations. They do not, however, provide an integrated way of seeing the details of computations or of constructing them.

Summary and future work

We have seen how large scale parallel computations can be designed and observed in ToonTalk by using the newly added ability to specify where houses should be built and how they should look. Currently, ToonTalk shows animations of a house being constructed or destroyed; future enhancements will show animations of trucks driving to the construction sites. By observing the trucks driving from the house that spawned a sub-computation to the empty lots where those computations will take place, we expect the causality of changes in the process structure to be more apparent. In some algorithms a single process spawns all the sub-computations, while in others, like the doubly recursive factorial, the spawning is done by recently spawned processes. Other patterns also occur. All will be observable when trucks are animated.

Similarly, birds are currently only animated coming into or leaving a house when observed from inside the house. We plan to animate the birds outdoors, enabling one to observe the communication patterns of parallel computations When doubly recursive factorial is executing one will see birds fly from houses to the houses one level up in the tree of houses.

One possible objection to this approach to animating large scale computations is that many relationships are invisible. For example, one might like to visualize processes with arrows displaying communication channels. In ToonTalk these channels are the relationship between a bird and her nest. The system could display them but it is hard to see how it could be done without interfering with the story behind the computation model of ToonTalk that is currently simple enough to be mastered by small children. An extension that might be useful that doesn't interfere would be to see from the outside of a house what activities are going on inside. Lights, for example, could be on when robots are working and dimmed if they are waiting or finished.

For very large computations one may need to have more levels of observation or abstraction. While a helicopter over a city enables one to see the computation at different scales, more is needed to understand large complex computations. We believe that the ability to have cities within cities could answer this need. While a bit odd, informal discussions show that children seem to be comfortable with the idea. After all, a child's toy model city is a city within a city.

A different way to perceive patterns in a computation is to augment the visuals with audio. Currently there are different sounds for each activity in ToonTalk. There is a sound, for example, of a house being built and another for a house being destroyed. There is no mixing of concurrent sounds, however. Instead, less important sounds stop when a more important sound starts. We plan to mix concurrent sounds – perhaps even decreasing the

contribution from events that are further away.

A good grasp of concurrent programming is an important part of a good computer science education. We hope that by providing tools like ToonTalk, intellectually significant and challenging computer science understanding and mastery can become "child's play".

Availability of ToonTalk and demos

A beta version of ToonTalk that runs on PCs with Microsoft Windows (3.1, 95 or NT) is available, as are the examples presented in this paper. For details send email to Kahn@CSLI.Stanford.edu.

References

[1] K. Kahn, "ToonTalk™ -- An Animated Programming Environment for Children", to appear in *The Journal of Visual Languages and Computing.* 1996.

[2] E. Shapiro, "Systolic Programming: A Paradigm of Parallel Processing", in *Concurrent Prolog – Collected Papers*, pages 2045-242, E. Shapiro editor, MIT Press.

[3] H. Kung, "Why Systolic Architectures?", *IEEE Computer **15**(1),* pages 37-46, 1982.

[4] H. Abelson and A. DiSessa, *Turtle Geometry*, MIT Press, 1981.

[5] S. Papert. *Children's Machines*. Basic Books, 1993.

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

[7] K. Kahn and V. Saraswat, "Complete Visualizations of Concurrent Programs and their Executions", in *Proceedings of the IEEE Workshop on Visual Languages*, pages 7-15, October 1990.

[8] E. Tick, "Visualizing Parallel Logic Programs with VISTA", in *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, Institute for New Generation Computer Technlogy, pages 934-942, Tokyo, June 1992.

[9] Y. Feldman and E. Shapiro, "Temporal Debugging and its Visual Animation", in *Proceedings of the 1991 International Symposium on Logic Programming*, V. Saraswat and K. Ueda editors, pages 3-17,MIT Press 1991.

[10] T. Disz and E. Lusk. "A Graphical Tool for Observing the Behavior of Parallel Logic Programs", *in International Symposium on Logic Programming*, pages 46-53. IEEE Computer Society, August 1987.

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

[12] K. Kahn and V. 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.

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