From Prolog and Zelda to ToonTalk

Ken Kahn
Animated Programs
49 Fay Avenue, San Carlos, CA 94070, USA
kenkahn@toontalk.com

This paper will appear in the Proceedings of the International Conference on Logic Programming 1999, edited by Danny De Schreye, MIT Press, 1999.

Abstract

ToonTalk looks like a video game. This is not surprising since its design and user interface were strongly influenced by games like *The Legend of Zelda: A Link to the Past* and *Robot Odyssey*. What may be more surprising is that ToonTalk is a programming language and environment based upon ideas that have evolved from Prolog over a period of nearly twenty years.

ToonTalk is a synthesis of ideas from concurrent constraint programming, video games, and programming languages for children. In the spirit of Logo [3], ToonTalk is an attempt to take the best ideas in computer science and make them accessible to children. When Logo was designed over thirty years ago, the best programming language ideas could be found in the Lisp language. The design of ToonTalk is based upon the belief that the best programming language ideas can be found in concurrent logic programming and concurrent constraint programming languages like Janus, Linear Janus, FGHC, Vulcan, DOC, AKL, and Oz [4,5,6]. These languages, in turn, borrow heavily from earlier languages like Concurrent Prolog and Parlog that in turn grew out of research on Prolog.

While these languages have many desirable aspects – they are powerful, elegant, theoretically well grounded, and expressive – they are not generally considered easy to learn. If it takes substantial time and effort for computer scientists to understand one of these languages, then how can one hope to make the underlying ideas accessible to young school children?

An answer lies with video games. Many of these games present a large and complex world with many kinds of objects and possible actions. And yet, children as young as 4 years old learn to master these game worlds without help. The fundamental idea underlying ToonTalk is that a game world can be created in which the objects and actions map directly onto programming language constructs. In ToonTalk, a clause becomes a robot, a term or tuple becomes a box, a number becomes a pad, and so on. The act of putting a box and a team of robots into a truck becomes a way of expressing a process spawn or procedure call. The act of dropping a number pad on another number pad becomes a way of expressing an arithmetic operation. And so on.

1 Concretizations

Concretizations are mappings between programming language abstractions and tangible objects. The idea is to preserve the semantics while providing a concrete analog for each computational abstraction. Ideally, the concrete analog should be a familiar object with widely known properties – e.g. boxes are good for holding things. Furthermore, these mappings work best when they fit together in a consistent theme – in the case of ToonTalk, a modern city was used. The ToonTalk concretizations are summarized in Table 1.

The ToonTalk programmer starts off flying her helicopter over a nearly empty city. When she lands she is followed by Tooly the Toolbox, which contains the essential building blocks and tools for programming. After entering a house, she can sit down and begin to train robots to work on boxes. She can then arrange for teams of robots to work in different houses, communicating by giving birds items to deliver to their nests.

Computational Abstraction	**ToonTalk Concretization**
computation	city
process	house
clause	robots
guard	contents of thought bubble
body	actions taught to robot
tuples or terms	boxes
comparison tests	scales
process spawning	loaded trucks
process termination	bombs
constants	numbers, text, and pictures
channel transmit capabilities (tellers)	birds
channel receive capabilities (askers)	nests
program storage	notebooks

Table 1 - ToonTalk Concretizations

A logic program is typically a collection of clauses. In concurrent logic programming languages the clause consists of a guard that specifies the preconditions for running the clause and a body that specifies what actions should be taken. In ToonTalk the guard corresponds to the box (i.e. tuple) that a robot is thinking about, which is displayed in the thought bubble of the robot. ToonTalk programmers understand that a robot will not work on a box unless it matches the box in his thought bubble. Matching is understood in visual terms. A robot is happy with boxes that have more detail than the one in his thought bubble, but he will not accept a box if something is in a hole of a thought bubble box and something else is in the corresponding location. The only exception to

this is if there is a nest in the corresponding location. A nest is the "ask" or read part of a variable. As with concurrent logic programming languages in this situation the arguments are insufficiently specified and the process is suspended. The "story" for ToonTalk programmers is that if a robot is expecting something and finds a nest instead, he does not give up. Instead he waits for a bird to come and cover her nest with something. If she covers her nest with the right kind of thing, the robot will wake up and start working.

The behavior of a robot is specified by the training he gets from the programmer. This training corresponds to defining the body of a clause. The possible actions are

- **sending a message** by giving a box or pad to a bird,
- **spawning a new process** 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 a process** by setting off a bomb, and
- **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 process. The last one may appear to introduce mutable data structures into the language, which are known to introduce complexity into parallel programs and make it difficult to provide a simple, clean semantics for the language. In fact, though, since boxes are copied and 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 a destructive operation is often a more convenient or direct kind of expression.

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 [2,7]. 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.

2 The *Append* predicate – an detailed example

Let us consider the classic *append* predicate of Prolog and concurrent logic programming. The programmer starts by connecting boxes together to construct sample input to the procedure. The box in Figure 1 corresponds to the arguments

in the call *append([a,b],[c],X)*. (The labels *List1*, *List2*, and *Both* are just annotations.)

Figure 1 – Sample input to Append

The programmer then gives the box to a fresh robot, illustrated in Figure 2A, and enters the robot's thoughts. As she does so, she stops controlling her persona and begins to control the robot, illustrated in Figure 2B. She starts by having the robot take the box out of the first hole and set it down.

A **B**

Figure 2 – Starting to train a robot to append lists

The program at this point corresponds to the following program fragment:

```
append(Arg1, Arg2, Arg3) :-
  Arg1 = [X1 | X2],
  X1 = a,
  X2 = [X3 | X4],
  X3 = b,
  X4 = [],
  Arg2 = [X5 | X6],
  X5 = c,
  X6 = [],
  Arg3 = $bird(Answer) | // guard is generated from the box in Step A
  Floor1 = Arg1, // from Step B
```

She then removes and puts the rest of the list back (step C), takes out a new nest to create a new asker/teller pair and drops the nest in the empty hole (step D), gives the box to the original bird in the box (step E), the bird then flies way and when she returns the programmer vacuums her up (step F), and drops the

bird that hatched from the nest in the hole (step G), and exits the robot's thoughts and labels him "App" (step H). Since she exited the robot's thoughts without setting off a bomb, the robot will repeatedly do what he was trained to do so long as the box continues to match the box in his thought bubble.

Figure 3 – Completing the recursive Append clause

The training of the robot is now complete and the corresponding program is:

```
append(Arg1, Arg2, Arg3) :-
  Arg1 = [X1 | X2],
  X1 = a,
  X2 = [X3 | X4],
  X3 = b,
  X4 = [],
  Arg2 = [X5 | X6],
  X5 = c,
  X6 = [],
  Arg3 = $bird(Answer) | // guard from the box given to the robot (step A)
  Floor1 = Arg1, // put contents of second hole on the floor (step B)
  NextArg1 = X2, // move contents of second hole back (step C)
  Floor2 = [X1 | Nest1], // drop nest in second hole (step D)
  Answer = Floor2, // give box to bird (step E)
  NextArg3 = $bird(Nest1), // replace old bird with new bird (steps F and G)
  append(NextArg1,Arg2,NextArg3). // recur since no bomb used
```

All that remains to finish this clause is to generalize it by removing details from the robot's thought bubble as illustrated in Figure 4. This effectively removes some of the constraints in the guard so that the guard becomes

```
append(Arg1, Arg2, Arg3) :-
  Arg1 = [X1 | X2],
  X1 = a,
  X2 = [X3 | X4],
  X3 = b,
  X4 = [],
  Arg2 = [X5 | X6],
  X5 = c,
  X6 = [],
  Arg3 = $bird(Answer) |
```

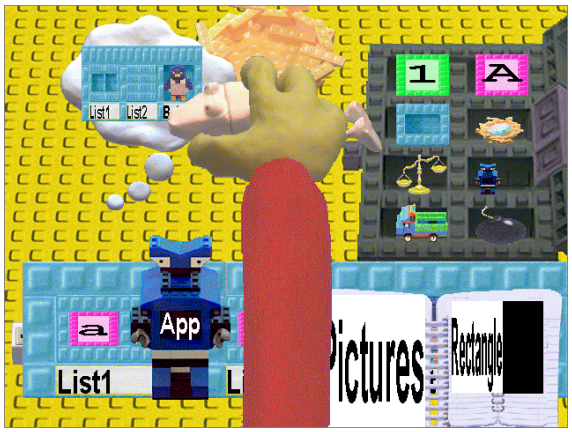


Figure 4 – The Append clause generalized (Step I)

After simplifying the clause by substitutions we obtain

```
append([X|Y],Z,$bird(Answer)) :-true |
  Answer = [X|Nest1],
  append(Y,Z,$bird(Nest1)).
```

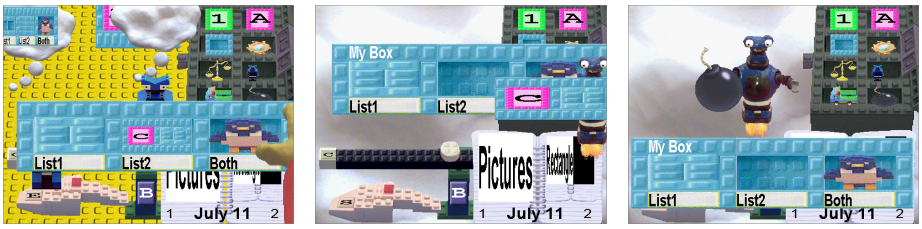
The programmer is now ready to define the base or termination clause. She gives the *App* robot the box and watches him execute two iterations before changing his box so that it no longer matches his thought bubble box as displayed in Figure 5K. The first hole of the box has a box with no holes (like [] in Prolog) while in the corresponding position in the thought bubble there is a box with two holes. The robot has filled the nest with the equivalent of the list [a, b | X].

J **K**

Figure 5 – Running the Append robot to generate the base case

The programmer gives the box to a new robot (step L) and trains him to grasp the box in the second hole and to give it to the bird (step M). Since in ToonTalk robots are implicitly recursive, she next overrides this default by taking a bomb and setting it off (step N). She then exits the thought bubble and removes the box in the second hole in the box in the robot's thought bubble. The resulting clause is

```
append(Arg1, Arg2, Arg3) :-
  Arg1 = [],
  Arg3 = $bird(Answer) |  // from thought bubble box
  Answer = Arg2 // give bird contents of the second hole (step L)
     .  // set off bomb(step M)
```



L **M** **N**

Figure 6 – Training another robot to terminate the computation

The programmer finishes by joining the two robots in a team and saving them in her notebook for future use. She can optionally add comments to the robots.

3 Discussion

ToonTalk is a new way of creating, testing, and debugging programs. One can think of ToonTalk as a very unusual animated syntax for a concurrent logic programming language and an integrated programming development environment that exploits the unique features of the syntax. Program elements can be learned and understood solely as ToonTalk "game" elements. ToonTalk programmers learn that birds take things to their nest and usually never learn that birds and nests are in fact send and receive capabilities on a communication channel. They can work out how their programs should behave by manipulating concrete sample values and only later abstracting their programs for generality. Simply watching their robots at work often is enough to understand a program or see a bug and how to fix it. This aspect of ToonTalk is a form of algorithm animation [1].

But a more global perspective often is necessary for a full understanding of a parallel algorithm. Inspired by the work of Shapiro on systolic programming in Concurrent Prolog [6], ToonTalk supports the ability to see the entire process structure of a computation. The programmer need only stand up, walk to her helicopter, and fly above the city as a computation runs. She sees houses being built (processes spawn), birds flying between houses (communications), and houses blowing up (process termination).

The default layout of the houses in a computation is that houses are built as close as possible to the house containing the robot that initiated the construction. One can override this default and create computations where stacks, trees, or grids of processes are laid out as houses that form lines, trees, or rectangular areas. The programmer can pan and zoom in and out of the unfolding computation by flying her helicopter around and up and down.

The process of designing an animated syntax for a concurrent logic programming language was not trouble free. Guards in ToonTalk are limited to what can be expressed as a pictorial matching process. Negation (e.g. a "not equal" predicate) could have been expressed by introducing some kind of visual

convention such as a circle with a diagonal line through it. But this would have added too much complexity. Instead, ToonTalk was designed so that teams of robots work sequentially. If the first robot of a team fails to match, then the robot directly behind him tries to match. This means negation can be expressed by the failure of previous robots to match. This diverges from the usual indeterminate committed choice clause selection of concurrent logic programming languages. In ToonTalk if there is insufficient information to decide the match the entire team suspends. This means that you can express the classic "merge" predicate of concurrent logic programming. Since ToonTalk communication channels (birds and nests) support many-to-one, one-to-many, and many-to-many communication this is not a critical shortcoming. Note that ToonTalk could be enhanced to have two kinds of teams: sequential and committed choice.

The earliest version of ToonTalk provided graphics, animation, and sound libraries via a message-sending interface in keeping with the spirit of concurrent logic programming. Testing with children and adults who were not computer scientists or software professionals revealed that this kind of interface is difficult to understand and master. The process of constructing a message and giving it to the correct bird in order to make a sound or move a picture was too indirect for most users.

To alleviate this problem a direct interface was provided based on the metaphors of remote controls and sensors. Some data objects are active in that they are repeatedly updated automatically. And some data objects when changed make a change to a corresponding element in a graphics or sound library. For example, a sensor for the width of a picture always displays the current width of the picture. If the sensor is changed, then the width of the picture changes accordingly. Children as young as 6 years old master these kinds of sensors and remote controls. But in a limited way, these constructs introduce shared mutable state to a concurrent programming system. To minimize the dangers of race conditions, sensors and remote controls in ToonTalk were limited to work in a very local fashion. But these constructs are a blemish from a theoretical point of view on the attempt to making concurrent logic programming widely accessible.

The computation model underlying ToonTalk is a concurrent logic programming language. Could other animated game-like worlds be constructed that matched other programming languages? It would probably be very difficult to construct a ToonTalk-like world for large complex languages, especially those that rely upon side effects to shared data like Java or C++. It would probably be much easier to make a ToonTalk-like world for other logic programming or constraint languages. For example, choice points could be "concretized" by splitting the ToonTalk city into multiple copies. A copy would be created whenever more than one robot in a team matches a box. The copies of the world would initially differ only by the choice of which of the matching robots gets to run. Cities that run into failure would just go away.

ToonTalk includes a Java applet generator. Anything built inside of ToonTalk can be exported as an applet that can run in a browser. Effort was expended to make the generated Java source code as readable as possible. A similar capability could be added to ToonTalk to produce source code in a concurrent logic programming language. The sample code presented here for the ToonTalk *Append* robots could have been automatically generated. Such a generator might provide a very nice "bridge" between the playful, game-like, child-oriented world of ToonTalk and the computer science underlying logic and constraint programming.

Acknowledgements

I am very grateful for the help, advice and support I have received from many people during the design and building of ToonTalk. In particular David Kahn, Mary Dalrymple, and Markus Fromherz deserve special thanks for all their help. Big thanks go to Ruth Peterson and her 4th grade class at Encinal School in Menlo Park, California where ToonTalk testing has been proceeding since 1995. And I am very thankful to the hundreds of beta testers throughout the world who have provided invaluable bug reports, comments, and suggestions.

References

[1] Marc H. Brown. *Algorithm Animation*. The MIT Press, 1987.

[2] Allen Cypher, Daniel C. Halbert, David Kurlander, and Henry Lieberman, editors. *Watch What I Do: Programming by Demonstration,* The MIT Press, August 1993.

[3] Seymour Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, New York, 1980.

[4] Vijay A. Saraswat. *Concurrent constraint programming languages*. Doctoral Dissertation Award and Logic Programming Series. The MIT Press, 1993.

[5] Vijay A. Saraswat, Kenneth Kahn, and Jacob Levy. Janus--A step towards distributed constraint programming. In *Proceedings of the North American Logic Programming Conference*. The MIT Press, October 1990.

[6] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 1989.

[7] David Smith, *Pygmalion: A Creative Programming Environment*, Stanford University Computer Science Technical Report STAN-CS-75-499, June 1975.