ARTICLE IN PRESS

Available online at www.sciencedirect.com

ScienceDirect

Journal of Visual Languages and Computing ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Journal of Visual Languages & Computing

www.elsevier.com/locate/jvlc

Towards a specification of the ToonTalk language

Leonel Morgado[a,*], Ken Kahn[b,1]

[a]*GECAD—Grupo de Investigação em Engenharia do Conhecimento e Apoio à Decisão, UTAD—Universidade de Trás-os-Montes e Alto Douro, Vila Real, Portugal*

[b]*Animated Programs, Oxford, UK*

Received 15 January 2007; received in revised form 5 October 2007; accepted 8 October 2007

Abstract

ToonTalk is a child-oriented programming language whose environment is an animated virtual world, with objects that children can pick up and use as in a game, such as birds, trucks, and robots, providing direct child-oriented metaphors for programming constructs. Actions performed by a programmer's avatar with these objects are both code and coding. ToonTalk is a powerful system, not just a "toy" system: it is based upon concurrent constraint programming languages, and programs written in languages such as Flat Guarded Horn Clauses and Flat Concurrent Prolog can be straightforwardly constructed in ToonTalk. However, there is not a specification of ToonTalk, for ready implementation in other environments. We propose that the ToonTalk language lies not in the animations displayed by the current environment, but on the actions performed by the programmer with virtual world objects; we present a description and analysis of the methods the ToonTalk language provides to programmers for expressing programs.

Keywords: ToonTalk; Animated programming; Action-based programming; Children programming; Concurrent programming; Virtual environments; Virtual worlds; Virtual environment

1. Introduction

ToonTalk [1,2] is the language that implements a novel idea in visual programming: *animated programming*. That is, using animation to express the code of a program, thus going one step beyond static visual programming. The commercial software product ToonTalk [3] is a proof-of-concept and commercial implementation of this language. It was the first language to employ this concept and is still the only known case (except for a few special cases in the physical world, as we will mention shortly), and was the base for several research projects involving the use of programming for educational purposes (for example, the Playground Project [4], the WebLabs Project [5], and doctoral research on programming with preschoolers [6–8]). Therefore, the presentation of the concept of animated programming, in the following paragraphs, is intertwined with the description and presentation of ToonTalk.

2. Animated programming

Being a novel concept, it helps to state what animated programming is not: it is not the

*Corresponding author. Tel.: +351 259350369; fax: +351 259350356.

E-mail addresses: leonelm@utad.pt (L. Morgado), kenkahn@toontalk.com (K. Kahn).

[1]Tel.: +44 1869331621.

1045-926X/$ - see front matter © 2007 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jvlc.2007.10.002

programming of animated characters or plays by means of static code; it is not the animation of the execution of static code; it is not the usage of animated icons or objects in place of traditional static icons.

Animated programming is the *usage of animation to express the code itself*. In order to help picture this concept, one can think of a programmer saying, "watch what I do" and of a system that records and interprets the visual (animated) actions of the programmer. This is just a simplification, for complete expression of code must also include methods for other programming constructs, such as specifying conditions and generalizing the actions of the programmer.

> The fundamental idea behind ToonTalk is that source code is animated (…). This does not mean that we take a visual programming language and replace some static icons by animated icons. It means that animation is the means of communicating to both humans and computers the entire meaning of a program ([2], p. 201).

In this regard, one can think of cases of "animated programming" in the physical world: for instance, when shooting a movie or rehearsing a play. Sometimes a director explains his/her intentions to an actor by performing actual body movements, gestures, facial expressions, etc. In this sense, there is "animated programming" of the actor's performance. By considering the physical world, some tangible programming systems [9], which employ physical objects to specify programs, can also be considered instances of animated programming, at least partially. For instance, the main feature of programming the Curlybot, "an autonomous two-wheeled vehicle with embedded electronics that can record how it has been moved on any flat surface and then play back that motion accurately and repeatedly" [10], is reproducing physical motion; and programming in Topobo, a "3-D constructive assembly system with kinetic memory" [11], is entirely done by turning the rotary joints of programmable pieces. (Yet, in these cases the "execution" of the program by the actor or robot is also an animation—an interpretation of the animation provided by the director/programmer. A generic animated programming system needs to go beyond mere cases of "animation to program animation".) However, programming a human actor, a Curlybot or a Topobo object all fall short of being general-purpose animated programming. For instance, while demonstrating something to an actor, a director often needs to resort to spoken speech to explain details that cannot be easily conveyed by movements of the human body or facial expressions; the Curlybot needs to resort to non-animated programming for constructs such as conditions and procedures; and Topobo programming is limited to the specification of motion for its programmable elements.

Furthermore, physical, tangible programming systems such as these are also limited in terms of resources (for instance, their bulkiness, the availability of enough programmable elements, and the cost of such elements). The specification or execution of a program may require 100 actors, or 100 Curlybots, for instance—a complex implementation situation. Their ability for general-purpose use also faces specific challenges at the level of user interaction. In this regard, tangible programming systems have been recently discussed in literature in terms of the Cognitive Dimensions (CDs) framework [12], traditionally used to assist the design and evaluation of graphical user interfaces. Specifically, they were analyzed in terms of tangible correlates to CDs, a proposed extension to CDs that aims to provide a generic framework for applying CDs to manipulable solid diagrams [13].

A virtual animated programming language, i.e., one based within a personal computer, not requiring physical props, can be limited only in the same ways as other programming languages (computability, processor performance, available memory, and hard disk space, etc.).

ToonTalk is such a language. It is based upon concurrent constraint programming languages, and programs written in languages such as "*Flat Guarded Horn Clauses, FCP, Parlog, Strand, and PCN can be straight forwardly constructed in ToonTalk*" ([2], p. 200), and vice-versa. As a consequence, ToonTalk is a powerful system, not just a "toy" system:

> We chose concurrent constraint programming as the underlying foundation of ToonTalk (…). One reason is that over ten years of use at many research centers has demonstrated that there is no risk that the language will be inadequate for building a wide variety of large programs. The languages are small yet very powerful ([2], p. 200).

At the level of user interaction, animated programming is an interesting case if paired alongside static visual programming or tangible programming.

Please cite this article as: L. Morgado, K. Kahn, Towards a specification of the ToonTalk language, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.10.002

As one would expect, several issues from tangible programming are not as relevant in (non-tangible) animated programming. For instance, the visibility cognitive dimension tangible correlate of the permanence of the notation, which in tangible programming systems raises significant challenges for some uses, as discussed in Ref. [13], is not as critical in animated programming systems, which can readily store and reproduce the animation notation. However, some circumstances in animated programming do seem relevant for an analysis from the perspective of tangible correlates of CDs. For instance, the ToonTalk user interface now includes a key for "unbamming", i.e., undoing the consequences of accidentally dropping an item on another (as we will describe further ahead, "dropping" is a central action primitive in ToonTalk animated programming), which may be analyzed under the tangible correlate of bulkiness. Regarding static visual programming, animated programming also presents specific issues for a CDs analysis. In terms of visibility, one cannot analyze what a program or program section does, for instance, without asking the system to reproduce it. And editing animated programming may present situations of high viscosity, as when a subprogram must be edited: it has to be reproduced until the point where the change is to be made, and in the current ToonTalk system that implies reproducing all actions in the subprogram from that change onwards, an aspect to improve. On other cases, viscosity is very low, as when one wants to change the kind of data a subprogram should work with, or when one wants to make a subprogram communicate with another, instead of simply producing an end result (all it takes is placing the head of a communication channel—a bird—where the end result would be placed, and possibly a quick erasing of an unnecessary constraint from a robot's thought bubble).

We believe that animated programming in general and ToonTalk in particular warrant a deep analysis of the cognitive dimensions of its user interfacing features, and we refer the reader to Ref. [13] for a possible starting point for such an effort. However, that is not our aim here: we simply clarify how the basic concept of using animation-producing actions to build computer programs can be and has been employed.

3. Why program with animation?

When ToonTalk was originally designed, research was already establishing that, regarding programming languages, "*visual representation can improve human performance*" ([14], p. 139). Several usability benefits were expected in ToonTalk from employing animation, some since demonstrated to be questionable or not as clear-cut as it was envisaged, but others since reinforced and clarified, as discussed below. The most important expectation was that animation could help users/programmers better understand the transitions between visual elements in what would be a visual programming language. Another expected benefit was that animation could help provide better metaphors, and consequently better mappings between programming concepts and those of users:

> (…) in order for a static picture to represent the dynamic behavior of a program, it needs to rely upon a rich set of encodings. Control and data flow need to be encoded in abstract diagrams. Abstract diagrams are arguably easier for people to deal with than symbolic formalisms, but they are still very difficult. Why not take the next step from visual programming languages and begin to use dynamic images, i.e., animation, to depict the dynamic processes of a program? ([2], p. 201)

Given the novelty of animated programming itself, there is no available research specifically regarding its use, by controlled comparison with static programming, to support or disprove these expectations, but informal favorable anecdotal evidence is mounting up (e.g., Ref. [4]). Research on some of the design principles adopted 15 years ago (and described 11 years ago in this journal) has also provided a better, albeit more complex, picture of the factors in play. We now know that the use of animations to convey abstract concepts may produce no significant benefit or even hinder learning in some cases [15,16], but that animation does help users to understand transitions in spatial data [17]. However, research has also shown that the way in which users interact with animations is an important factor: manipulation animations have different cognitive impacts from simulation animations. The former have an enabling function: they enable users to perform more cognitive processing than with static pictures; the latter have a facilitating function: they support matching mental processes, rendering them easier, but this sometimes leads to negative overall effects [18]. There is also a better understanding of how different people are affected by these kinds of animation: for instance, those with high learning prerequisites seem to be mainly

Please cite this article as: L. Morgado, K. Kahn, Towards a specification of the ToonTalk language, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.10.002

affected by enabling animations and their benefits, whereas those with low learning prerequisites seem to be more subject to the negative effects from facilitating animations [18]. In ToonTalk, the user/programmer is producing the animations by manipulation, not simply being presented with the simulation of processes; thus, ToonTalk's use of animation fulfills a mainly enabling function. Also, ToonTalk users/programmers can control the pace and order in which the animations are created, and are actively involved in the process of understanding. Further, the programming environment generates feedback and support during the programming process, involving the user in the active understanding of the events taking place, as we describe further ahead—features that are now recommended for effective use of animations ([16], p. 264).

Regarding the importance of metaphors in visual programming languages, research has since shown that—at least with adult users—employing visual metaphors is useful, but "the effect of metaphors is smaller than either the effect of expertise or the effect of pictorial mnemonics" [19]. ToonTalk's animated metaphors for programming constructs are also rich in pictorial content, however, and extendable by the use of pictures. An unforeseen positive consequence of this was that ToonTalk allowed over 100 preschool children (aged 3–5) and their teachers to make computer programs employing a larger variety of programming techniques than it had been hitherto possible with other programming languages for preliterate children, including parallel/concurrent statements, message passing, input guards, and clients and servers [8].

4. Issues regarding implementation and description of animated programs

In order to provide a description of specific features of ToonTalk, we need to address two problems with the implementation and description of animated programs: the *creation of animation* by the user/programmer and *providing a static description* of the programs (for instances such as their presentation on printed matter).

The creation of animation is a basic problem for any animated programming language: since the source code itself is animated, the user must be able to produce the required animations. But producing animations is often a complicated task in itself. The solution provided by ToonTalk is to make the programming environment resemble a video game, where the user controls and manipulates game elements while programming.

> (…) 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 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 ([2], p. 201).

Even considering that the animation is that of a traditional video game, the other aforementioned problem remains: animated code, be it ToonTalk code, filmed gestures or some other situation, is troublesome to describe in a static medium such as this written document. Sometimes a simple screenshot will suffice, but not always. Circumventions for this have been used, and throughout this document we will employ the approach of capturing pieces of the action as still frames and arranging them in comic-book style (Fig. 1). This approach has been used previously, and such pieces have been called "snapshots" [2] or "comic strips" [20].

In some circumstances, however, the lack of motion in these images may pose doubts to the reader. In those cases, we add textual comments below each frame. These comments will convey to the reader some of the information unavailable due to lack of motion.

5. The ToonTalk programming environment

As mentioned above, currently ToonTalk is the name for both a programming language, and a programming environment where that language is used to create programs. The most striking feature of the programming environment is that it is child-oriented, and for this reason there is a strong emphasis on providing a global metaphor for conducting programming activities. As much as possible, all the programming activities take place in a video-game world resembling a town or

Please cite this article as: L. Morgado, K. Kahn, Towards a specification of the ToonTalk language, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.10.002

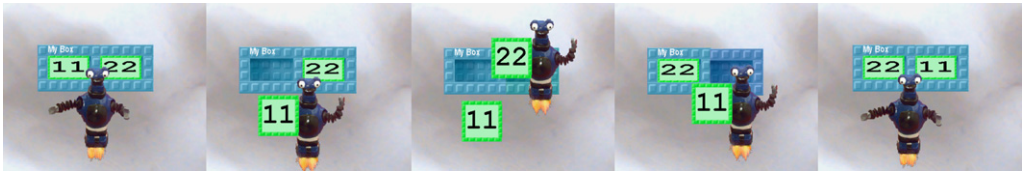


Fig. 1. Snapshots from swapping two elements (from Ref. [2], p. 202).

Fig. 2. ToonTalk programming environment: programmer and toolbox outside.

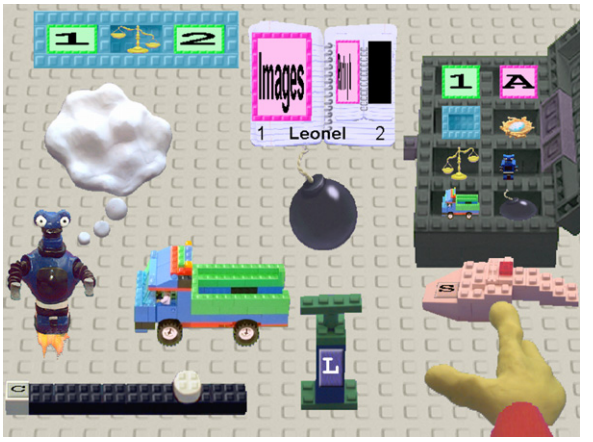


Fig. 3. ToonTalk programming environment: programmer's hand and toolbox contents.

suburb—which in ToonTalk terms is called a "city". The user (programmer) controls a doll/figurine, seen as his/her programming persona—the programmer's avatar (Fig. 2).

Controlling that avatar, in video-game fashion, the user can enter and leave houses, where computations are taking place, observe and debug ongoing computations, and create new computations using objects representing tools and primitives, stored inside a legged, trailing, toolbox. Since the city can grow large, a helicopter is provided, to allow a bird's-eye view and simplify navigation in the city (the toolbox and the helicopter are visible in Fig. 2).

The programming takes place within this global metaphor. By clicking the mouse the programmer "kneels" on the floor of a house, to program or debug. This makes the toolbox open, displaying the basic programming primitives: number pads, letter pads, and other "physical" objects, which can be used for performing comparisons, process[2] spawning, process termination, etc. The metaphor is maintained because now the mouse movements still control the programmer's avatar, visible as a hand and an arm (i.e., not just a cursor shaped as a floating hand). Fig. 3 presents this hand, the open toolbox on the floor, and several tools and primitives.

By "tools", we are referring to elements whose only purpose is to allow the programmer to manipulate the environment and the visual elements used in programming: producing copies, zooming in and out, deleting (vacuuming), erasing (creating a generic version of an element), saving to disk, etc. They are extensions to the programmer's avatar. All other elements of the ToonTalk programming environment are primitives of the ToonTalk programming language. In this taxonomy, the programmer's hand and the programmer's avatar are not classified at all (neither as tools nor as primitives); they are meant to be seen as extensions to the human programmer's physical actions. One should have in mind that since tools are extensions to the avatar, a different "programmer" might not need them to achieve the same purpose: for instance, if the programmer's avatar was a winged dragon, rather than a human, there would be neither a need for an helicopter to fly around, nor for a vacuum cleaner to get rid of things (albeit charring things down to the ground might

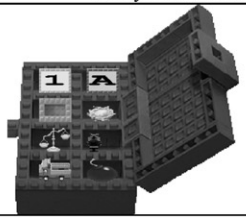
[2]The term "process" is used here just for the sake of simplicity in this short description. "Agent", "actor" or "object" could also have been used.

Please cite this article as: L. Morgado, K. Kahn, Towards a specification of the ToonTalk language, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.10.002

not be a good idea should one need them later). In this sense, the elements which we classify as ToonTalk tools are the ones presented in Table 1.

An alternative computer science perspective could be to disregard the existence of tools altogether, since all their purposes are potentially replaceable

Table 1
ToonTalk programming environment tools and their purpose

ENVIRONMENT TOOLS			
Visual (when not used)	**Visual (in use)**	**Modes of operation[a]**	**Location**
Dusty the Vacuum			
		S – Suck up element R – Regurgitate E – Erase surface	Inside Tooly the Toolbox. Pops out when the programmer kneels.
Helicopter			
		Flying	Landed on city streets. Can be summoned with the "F1" or "H" keys.
Maggie the Magic Wand			
		C – Copy element S – Copy itself O – Copy original picture of an element	Inside Tooly the Toolbox. Pops out when the programmer kneels.
Marty the Martian			
		Providing help with text balloons Providing help with audible speech	Shows up whenever the programmer is inside a house. Can be sent away or summoned with the F1 key.
Pumpy the Bike Pump			
		B – Make big L – Make little W – Make wide N – Make narrow T – Make tall S – Make short G – Make "good" size	Inside Tooly the Toolbox. Pops out when the programmer kneels.
Tooly the Toolbox			
		Following the programmer Open on the floor or in hand	Trails behind the programmer when he/she is walking or standing up. When the programmer kneels, Tooly opens up.

[a]The modes represented by letter buttons are presented as they appear in the US English version, but are found localized in other versions of the ToonTalk programming environment.

Please cite this article as: L. Morgado, K. Kahn, Towards a specification of the ToonTalk language, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.10.002

by a programmer's avatar: in the previous paragraph, we mentioned the use of a winged dragon avatar as a way to do away with the helicopter and at least part of the vacuum cleaner functionality, and this line of reasoning can be applied to the other tools (for instance, the magic wand would be unnecessary if the programmer's avatar was a creature able to produce copies of anything it ate, such as a Grumpy Converter from Bluxte [21]). As we describe in the next section, the syntax of ToonTalk is not dependent on the tools themselves but on the primitives, in the form of physical objects, of actions between the user and the objects, or of actions amongst objects (all these actions are visible through user-controlled or automated animations). Conceivably, any animation displaying an action can be replaced by a completely different one, as long as its effect and applicability remain identical, and the language would still be ToonTalk, even if the overall appearance changed a lot: "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" [2].

All elements in the ToonTalk programming environment are "cartoons", even the tools. In line with the language and environment metaphors, they possess animations. Table 1, which details the tools, presents examples of such animations.

6. The primitives of the ToonTalk programming language

Aside from those visual elements classified as "tools" in the previous section, all remaining visual elements in the ToonTalk programming environment are primitives of the ToonTalk programming language. But by "visual elements" we are not referring solely to the physical objects: they do not constitute the entire set of primitives. As expected in an animated language, several primitives are animated actions, not just physical objects. These two kinds of primitives are *object primitives* (the physical objects, which can be seen as values), and actions upon them or amongst them, visible through user-controlled or automated animations, which we call *action primitives* (and can be seen as operations). Together, they give the programmer the power to express a program in ToonTalk. Table 2 lists the object primitives and Table 3 the main action primitives available in ToonTalk.

The aforementioned tables present detailed lists of ToonTalk environment tools and language primitives, but they still fall short of providing an actual picture of its programming. A crucial element is the behavior of a specific primitive, the "dropping on" primitive. As described in Table 3, this "dropping on" primitive represents the actual action of taking an object primitive and releasing it so it falls on top of another object primitive. But what is the result of this action? As stated in Table 3, "*the actual operation depends on the nature of the primitives being combined*". In the visual example included in that table, for instance, a number pad for "1" was dropped on another number pad with a "1", and the result was that those numbers were added, resulting in a single number pad, with the number "2". However, for other primitives, the result can be quite different: for instance, dropping an array on a robot without constraints results in the initiation of the programming of the operations of that robot. (This particular case is a basic building block of ToonTalk programming.) Yet other combinations of primitives produce results such as sending a message, spawning a process, or defining a template. For this reason, we provided Table 4, which details the results of the combination of different object primitives, by dropping one on top of another.

7. Doing the coding of ToonTalk programs

Fig. 1 presents an example of a program performed using the primitives and combinations just presented (an "exchange two numbers" program). An example of a running program, using two robots that communicate amongst them is presented further ahead, on Fig. 12.

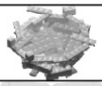
In language-specific terms, a ToonTalk program is built by programming robots. This programming is achieved by executing a sequence of actions such as those in the following example, describing the process of coding the program in Fig. 1, for exchanging two numbers:

1. Create a cubby box with a starting example: elements that define the robot's constraints (Fig. 4).

Please cite this article as: L. Morgado, K. Kahn, Towards a specification of the ToonTalk language, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.10.002

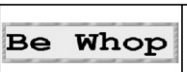
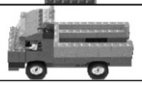
2. Drop that array on an untaught robot (Fig. 5).
3. Upon entering the robot's thought bubble, use the mouse to control the robot, and perform the desired set of actions on the constraints (see Fig. 1).
4. The result is a robot with a set of constraints on its operation (Fig. 6).
5. The constraints of this robot can be generalized [22], allowing it to work on any numbers or even

Table 2
ToonTalk object primitives

OBJECT[a] PRIMITIVES		
Primitive	**Visual representation**	**Purpose**
Bird		Message sending.
Bomb		Termination of a set of programming elements (see row on "house") or of a programmable object (see row "programmable object")
Programmable object	See rows on "Image", "Text pad", and "Number pad".	Holding a set of programming elements, running concurrently with access to common properties such as "distance from the left side", "speed to the right", "collision state", or even other object primitives. All images, text pads and number pads are programmable objects[b].
Cubby box		Organization of elements in arrays.
House		Holding a set of independent programming elements. In computer-science terms, it can be seen as an object, a process, an actor, etc.
Image		Representation of graphical data.
Infinite stack	Identical to the object primitive used in its creation.	Template for instantiation of an object primitive.
Nest		Message reception. First-in, First-out container of received messages
Notebook		Organization of elements in collections[c] of templates.
Number pad		Representation of numerical values (static data or otherwise), as numerical constants, variable values, or arithmetic expressions. See also the row on "programmable object".
Robot		Representation of a sequence of operations (*i.e.*, a procedure).
Robot thought bubble		Representation of constraints on the operation of the robot.
Robot team		Prioritized set of robots (i.e., a like a "case" decision structure): the constraints of the first robot in the set are tested first; only if they fail will the constraints of the second robot be tested, and so on.
Scale		Representation of the result of comparisons (static to specify a result, animated to specify indetermination).

Please cite this article as: L. Morgado, K. Kahn, Towards a specification of the ToonTalk language, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.10.002

Table 2 (*continued*)

Multimedia pads (sound pad, force-feedback pad)		Representation of audio (both sound data and text-to-speech) or multimedia effects (currently, only joystick force-feedback).
Text pad		Representation of alphanumerical strings (static data or otherwise). See also the row on "programmable object".
Truck		Spawning of new sets of programming elements (see table row on the object primitive "house"). Container for primitives related to spawning.

[a]Most of these primitives are not visually static, they possess animations. However, those animations do not represent actions and serve no specific programming purpose, and are employed solely for aesthetic or metaphoric reasons.[b]This is the current status of the ToonTalk language and environment, which does not possess an actual specification document. Envisioning such a specification, one could consider all object primitives as programmable objects, and the current ToonTalk software simply as a language implementation where only some of those programmable objects are usable as such.[c]The term "collection", used here, aims to convey a distinction between notebooks and arrays (which are represented by cubby boxes). While ToonTalk collections are ordered (i.e., notebooks have page numbers), just like arrays, they are distinguishable from ToonTalk arrays in several aspects. Firstly, they are indexed: in a notebook, a program can directly access an element, using a value that only becomes defined at runtime (either by page number or textual or pictorial content of pages); cubby box arrays must be manipulated or iterated. Secondly, notebooks do not have a defined size: they hold an unlimited number of elements, but the only way to specify a number of "empty" cells/pages is to have a non-empty cell after them in numerical page order. Thirdly, notebooks are not divisible: there is no primitive to split the contents of one notebook over two notebooks, whereas with cubby boxes one can readily do that operation.

any other array contents. This is achieved by erasing the number pads (meaning "any number") or vacuuming them (the empty-hole constraint means "any content"), as can be seen in Fig. 7.

Such robots can be activated by dropping on them an array with which to work. For instance, a robot such as the one on the right-side in Fig. 7, with generalized constraints, will accept any of the arrays in Fig. 8, but not the arrays in Fig. 9.

In Fig. 8, all the cubby boxes have a number on the first hole, matching the first-hole constraint of the robot in Fig. 7 ("any number"). The contents of the second hole do not matter, because there is not any constraint on the contents of it.

In Fig. 9, the first and second arrays do not have numbers in the first hole, and so do not match the first-hole constraint. The third array does have a number in the first hole, but the box is a three-hole box, and the robot's thought bubble constraint on box size specifies a two-hole box. Table 5 presents a summary of these matching rules. A point to note in matching is that labels, comments, and names are never considered. In ToonTalk, all these are simply provided for the convenience of the programmer (or anyone curious about what the program does), and do not constitute programming elements (i.e., one can neither lookup a bird named "Polly", a box labeled "X", nor a notebook labeled "Samples").

8. Debugging support in the programming environment

The ToonTalk programming environment provides visual cues to help the programmer detect constraint problems such as those described above. When a constraint is not fulfilled, the robot stops and that failing constraint is highlighted in red (Fig. 10).

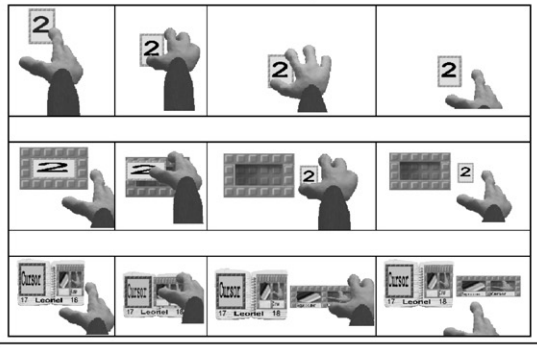
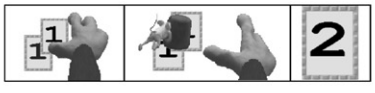
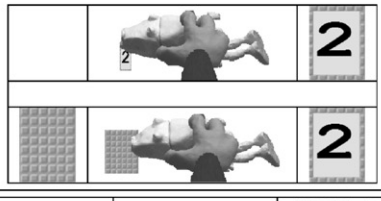
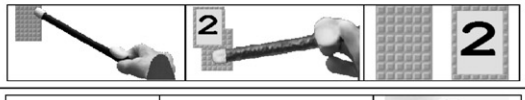
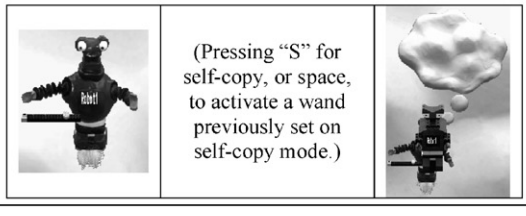
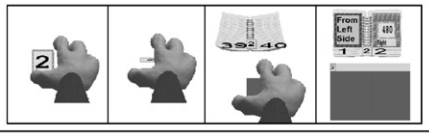
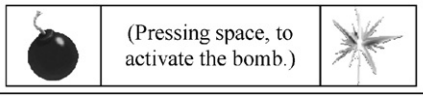
These visual cues are one of the features of the ToonTalk system that help the programmer understand the program execution. But perhaps the most important feature on this regard is the possibility of observing the execution of a program at various speeds and levels of complexity, as we will now explain.

After programming a robot, there are four different ways to make it execute its sequence of operation. These not only provide for different program requirements, but also help the programmer understand the execution of the program. They are:

1. Put the robot on the floor and directly drop on the robot the box for it to work with.
2. Put in a truck both the robot and the box with which it will work (the truck will do the dropping-on, inside a new house).
3. Put the robot on the back of a picture and directly drop on the robot the box for it to work with. The robot will start working when the picture is flipped over.

Please cite this article as: L. Morgado, K. Kahn, Towards a specification of the ToonTalk language, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.10.002

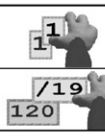
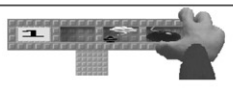
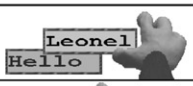
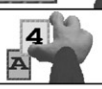
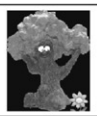
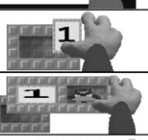
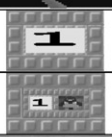
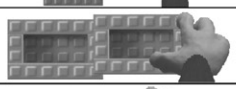
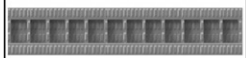
Table 3
ToonTalk main action primitives

MAIN ACTION PRIMITIVES		
Primitive	**Animation snapshots**	**Purpose**
Grabbing and dropping		Repositioning of object primitives – OR – removal from a programmable object, cubby box, or other container – OR – instantiation of an object primitive from a template.
Dropping on		Combination of object primitives. The actual operation depends on the nature of the primitives being combined. Not all combinations are valid.
Vacuuming		Elimination of an object primitive.
Erasing		Generalization of an object primitive. Not applicable to all object primitives.
Spitting Out		Recovery of an object primitive that was recently vacuumed – OR – recovery of a defined state from a recently generalized object primitive.
Magic wand copy		Duplication of an object primitive.
Magic wand copy of original		Creation of a duplicate with a defined state, from a generalized primitive.
Magic wand copy of self	 (Pressing "S" for self-copy, or space, to activate a wand previously set on self-copy mode.)	Creation of a duplicate of the current process.
Flipping a programmable object		Access to its contents, for addition, removal, commenting, and labeling.
Setting off a bomb	 (Pressing space, to activate the bomb.)	Termination of a set of programming elements or of a programmable object

4. In any of the three previous methods, replace some of part of the box contents by nests; the robot will only start working when object primitives are eventually dropped on those nests.

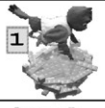
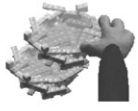
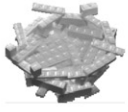
The *first method*, presented in Fig. 11, is a typical test (or debugging) situation. Upon being given a box matching its constraints, a robot will proceed with the execution of its sequence of operations, under the gaze of the programmer, at a speed

Table 4
Main results in ToonTalk of using the "drop on" primitive

"DROP-ON" COMBINATION OF OBJECT PRIMITIVES – Main results			
Involved primitives	**Visual combination**	**Visual result**	**Result description**
Number pad on number pad			Result of the arithmetic operation. Default operator is "+".
Cubby box on number pad			The array is split in two: the first *n* elements in one (*n* is the value of the number pad) and the remaining elements in another.
Cubby box on erased number pad			Number pad with the number of items in the array (number of cubby boxes).
Text pad on text pad			Concatenation of text strings.
Number pad on text pad			The letter index is increased.
Number pad on generalized ("erased") text pad			Alphanumeric representation of the numerical value in the number pad.
Image, number pad or text pad on image			The dropped-on image or pad becomes an independent part of the bottom image.
Image on erased image			The bottom image acquires the visual looks of the dropped-on image.
Object primitive on cubby box			Placement of the dropped-on primitive inside the cubby box.
Cubby box on side of cubby box			Concatenation of the two arrays.
Text pad on erased cubby box			Text converted into an array with a single-letter pad in each hole.
Number pad on erased cubby box			The erased box becomes an array with as many empty boxes as the dropped-on number.
Notebook on erased cubby box			The notebook templates are instantiated. Each element is placed in sequence in the resulting array.

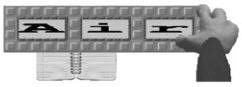
Please cite this article as: L. Morgado, K. Kahn, Towards a specification of the ToonTalk language, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.10.002

Table 4 (*continued*)

Robot team on erased cubby box			The erased box becomes an array with as many holes as there were team members. The robot team is split, and each member is placed into a hole.
Object primitive on bird			The bird drops the object primitive (and copies of it, if necessary) on all its associated nests[a].
Object primitive on empty nest	 		Placement of the dropped-on primitive inside the nest.
Object primitive, carried by bird on non-empty nest			Placement of the dropped-on primitive inside the nest, below[b] existing contents.
Nest on empty nest			The two nests are joined. Birds associated with either nest become associated with the resulting nest.
Object primitive on flipped programmable object			Placement of the object primitive inside the programmable object.
Cubby box on untaught robot			Change of setting to the robot's thought bubble; until exit from this setting, all action primitives are recorded as that robot's sequence of operations. The status of the dropped-on array is recorded as that robot's constrains.
Cubby box on taught robot			The robot's constraints are compared to the dropped-on box. If they match, the robot executes its sequence of operations repeatedly, until the constraints cease to be valid.
Cubby box on vacuumed thought balloon of taught robot			Assignment of new constraints to the robot's sequence of operations.

Please cite this article as: L. Morgado, K. Kahn, Towards a specification of the ToonTalk language, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.10.002

Table 4 (*continued*)

Cubby box on taught robot whose thought balloon has been vacuumed			Retraining. The setting changes to the robot's thought bubble, where the robot replays its sequence of operations, either until the end or until the programmer interrupts it, by clicking a mouse or keyboard button. Any subsequent actions are recorded following the replayed ones. (Unreplayed actions are discarded.)
Robot on robot			Robot team. From the viewpoint of all other primitives, a team is a single robot. Within the team, if a dropped-on array fails to meet the constraints of a team member, the array is passed on to the next team member in line.
Robot AND cubby box on truck (in any order)			Spawning. A new house is created. The truck drops off the robot there and drops the array on the robot. Other things such as address sensors, house pictures, and notebooks can also be dropped on the truck (before either the robot or the box) to define various details[c] of the spawning.
Object primitive on empty page of a notebook			The object primitive becomes a template, stored in that page of the notebook.
Number pad on non-empty page of a notebook			The notebook opens on the page matching the dropped-on number.
Text pad on non-empty page of a notebook			The notebook opens on the page whose textual contents match the dropped-on text.
Cubby box on erased notebook			The array elements are converted into templates, one in each page of the resulting notebook.

[a]See Section 11 for more details.[b]The picture in the "visual result" column was edited for the benefit of printed clarity. In ToonTalk, the presence of items "behind" the first is not visible until the first one is removed.[c]The address sensors can hold the address of the city block where the house should be created (it will be created in the first available lot within that block). A notebook dropped on a truck means that the robot will use it as its own main notebook, instead of the programmer's, thus avoiding contact with a global state. House pictures will determine the appearance of the new house (if no pictures are used, it defaults to the appearance of the house where the spawning was performed).

Please cite this article as: L. Morgado, K. Kahn, Towards a specification of the ToonTalk language, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.10.002

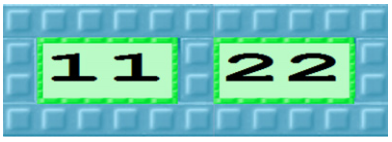


Fig. 4. Starting example for a sample robot.

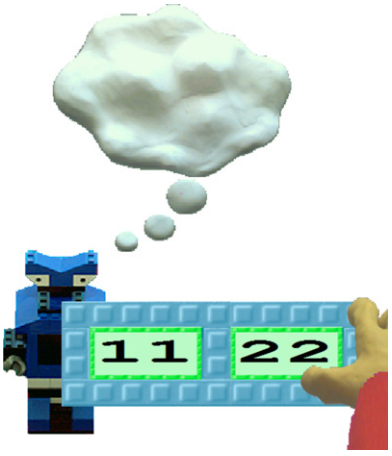


Fig. 5. Dropping on a robot an array with the starting example.

Fig. 6. Taught robot, thought bubble identifies constraints.

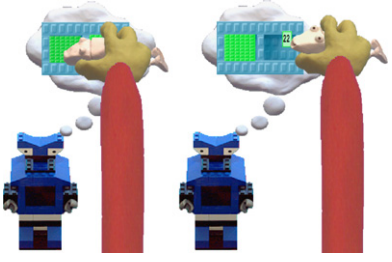


Fig. 7. Generalizing constraints in ToonTalk: erasing and vacuuming.

identical to that used by the programmer to perform the same actions (albeit without pauses).

This method allows one to finely observe the behavior of a robot with a specific array, to determine whether its actions are the intended ones. However, since the execution proceeds at human pace and with the robots in view, it is usually not suited to most execution situations. But this is not a real limitation, because the programmer's avatar can get up, and leave the house where the robots are working. By doing so, the robots will execute at full speed. Therefore, this method can be suitable for all situations in which the visual presence of the robots does not place a problem. The programmer can let the computation proceed at full speed inside the houses, and simply enter a house to see the robots working in detail. There are three levels of detail and execution speed for such observations: while the avatar is kneeling, the robots execute with full detail, at a speed similar to that of a human programmer; when the avatar is standing up, the detail level is reduced and the execution speed is increased; when the avatar leaves the house and flies over the city in the helicopter, the level of detail if further reduced, displaying only trucks (spawning), bombs (termination), and birds (communication), no finer detail.

The *second method* means that the moment the truck departs, the computation is running, but not under the eye of the programmer. This is an adequate situation in several cases. For instance, if one already debugged the program reasonably well and simply wants it to execute and terminate autonomously; or if one wants it to run continuously and simply observe the computation results indirectly, as messages being delivered to a programmable object or some other remote effect (changes to images in view for example).

The *third method* is similar to the second, in that the robots will be executing at full speed, and the programmer can only observe indirect effects. Its difference lies in practical issues. For instance, if one wants to inspect all robots manipulating an image's properties, it is easier to find them stored behind that image, rather than having to determine which houses hold them, amidst many others. And the

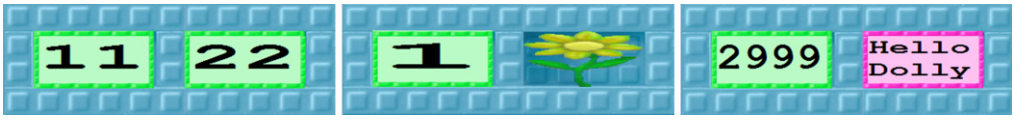


Fig. 8. Arrays acceptable for the robot in Fig. 7.

Please cite this article as: L. Morgado, K. Kahn, Towards a specification of the ToonTalk language, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.10.002

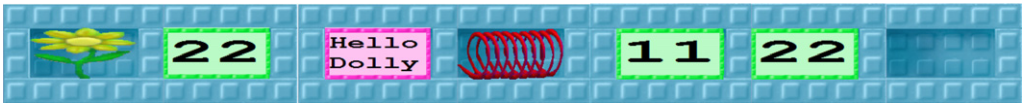


Fig. 9. Arrays unacceptable for the robot in Fig. 7.

robots can use the picture's sensors to provide some feedback regarding their state (e.g., changing its position, size, color, etc.). Also, sets of robots working behind pictures can be placed as a group behind yet other pictures, or stored together in a notebook, along with their assigned boxes. This simplifies the combination and reuse of different programs. Finally, a new set of robots can be activated rapidly, by simply creating a copy of a picture with robots behind (by copying an existing picture or by instantiating a template), rather than having to spawn a new house for each robot in a group.

The *fourth method*, as stated in its description above, is a modification to the three previous methods, based on the technique employed in ToonTalk programming to perform robot-to-robot communication: sending messages through birds. This method consists in providing a robot with a box whose contents do not entirely match the robot's constraints: rather, some of the box's holes will contain only a nest. Such a robot, when about to start working, will simply be idle, until the nests in its box have received objects that match the robot's constraints. In Fig. 12, for instance, the robot on the right is working, but doing nothing: it is waiting until something arrives in the nest, for then to check it against its constraints (in this case, it has to be a number). The nest belongs to the bird on the left. The left-side robot, called "Adds 1", has been taught to increase its left-side number by 1 and drop a copy of the result on the bird. The bird will take it to the nest in the right-side robot which will then proceed.[3]

An example can clarify the way in which this technique may also be used to help the programmer understand the behavior of a program. The robot on the right side in Fig. 12 accumulates the received values in the hole labeled "sum". Supposing that the number it holds is not just any number, but a control for a picture's width, then running this robot on the back of a picture (while the other robot from Fig. 12, running in another house, provides numbers), will simply make it grow large very fast (Fig. 13), because the left-side robot in Fig. 12 will provide new numbers quickly and the right-side robot in Fig. 12 will update the picture at an identical speed.

Should the programmer find this behavior odd, he/she may want to inspect the impact of the width-changing robot. But waiting for its animation to unroll for each received number may be tedious. Using the bird and nest can be helpful. The robot can stay behind the image, working on its width, and the programmer can hand new numbers to the bird directly, watching the resulting effect on the picture width, rather than having to wait for the animation of the robot (Fig. 14).

9. Linking to the machine state of the programming system

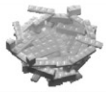
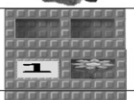
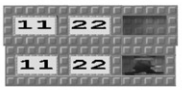
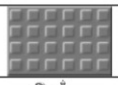
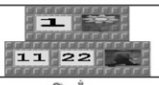
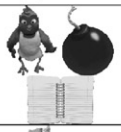
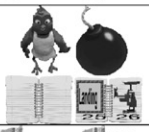
The manipulation of controls for a picture, as mentioned above, is perhaps easier to understand if detailed a bit further. The "controls" themselves are regular object primitives (text pads, number pads, boxes, images) that present special behaviors. This is how ToonTalk links itself to the state of the programming system. Terminology-wise, these object primitives with special behaviors are only called "controls" when they provide access to the state of an internal ToonTalk programmable object (a picture, for instance); they are called "sensors" instead, if they provide access to the state of the computer system or the overall ToonTalk programming environment (mouse position, pressed keys, looks of the current ToonTalk house, etc.).

The sensors can be found in the "Sensors" notebook, within ToonTalk's main notebook. The controls, being specific to a programmable object, can be found in the back of each programmable object, which can be accessed in several ways, while the programmable object is being held: pressing "F" for "flip"; holding down a shift key and left-clicking

[3]This pair of robots produces the following results: on the left, at each iteration the number will increase by one, so the number will be 0, 1, 2, 3, 4, …; on the right, the number dropped on the nest is added to the value already in the hole labeled "sum", which will be "0, 1, 3, 6, 10, …", i.e. the result of $0+1+2+3+4+\ldots$

Please cite this article as: L. Morgado, K. Kahn, Towards a specification of the ToonTalk language, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.10.002

Table 5
Summary of constraint-matching rules in ToonTalk

SUMMARY OF CONSTRAINT-MATCHING RULES IN TOONTALK			
Constraint in thought bubble		**Valid matches (in boxes handed to robot)** Note: sensors match as the value they are currently "portraying"	
Any constraint but empty nests.		An empty nest is not checked to see if it matches; rather, the checking is suspended until an object arrives in the nest, and that object is then checked.	
Empty nest		Empty nest.	
Ungeneralized pads (text, number, sound)		Pads which present the same exact values (the contents on the back are not checked). Numerical formats do not affect the matching (e.g., 1.5 matches 1.5, 1½ and 3/2).	
Generalized pads (erased pads)		Same-type pads regardless of appearance or contents on the back (the type of the pad is indicated by its colour).	
Ungeneralized images		Images whose base pictures are identical, regardless of width and height (the contents on the back are not checked).	
Generalized images (erased images)		Any image, regardless of the appearance or contents on the back.	
Empty boxes		Boxes with the same number of holes (regardless of contents).	
Boxes with contents		Boxes with the same number of holes, where the matching applied to each hole is valid.	
Mixed boxes (some holes filled, some empty)		Boxes with the same number of holes, where the matching applied to each hole is valid (any content is valid for the empty holes).	
Generalized boxes (erased boxes)		Boxes with any number of holes, regardless of content.	
Scales		Scales in the same state (balanced, left-tilting, right-tilting, or wobbling)	
Robot or robot team		Any robot team whose first members are copies of the same robots, in the same sequence, regardless[a] of their names, constraints, or secondary members of the team. (Robots and teams can be erased to match any other robot or team.)	
Birds, bombs, or notebooks		Respectively, any bird, bomb or notebook (notebook contents and state are not matched).	
Empty trucks		Any truck, regardless of contents.	
Trucks with cargo		Trucks whose cargo matches.	

[a]In this row, the rightmost cell shows two sample matching teams. Firstly, a team that is identical to the constraint. Secondly, a team where several changes occurred, but that still matches the constraint: a new robot was added to the team, but since it is located behind the first two, it is not considered for matching purposes; and the constraint of the second robot was changed, but the robot itself is the same.

the mouse; and, if the user activates the appropriate program customization option, right-clicking the mouse. Fig. 15 presents the resulting animation, with a picture (a programmable object) being flipped to reveal its contents (in this case, it is empty) and the notebook of controls flying out of it.

Please cite this article as: L. Morgado, K. Kahn, Towards a specification of the ToonTalk language, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.10.002

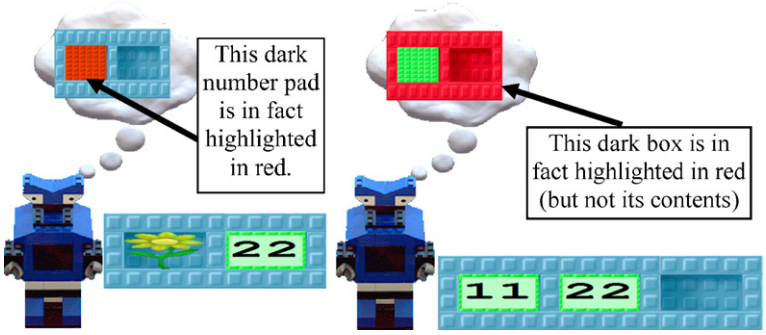


Fig. 10. ToonTalk robots after constraints failed to match.

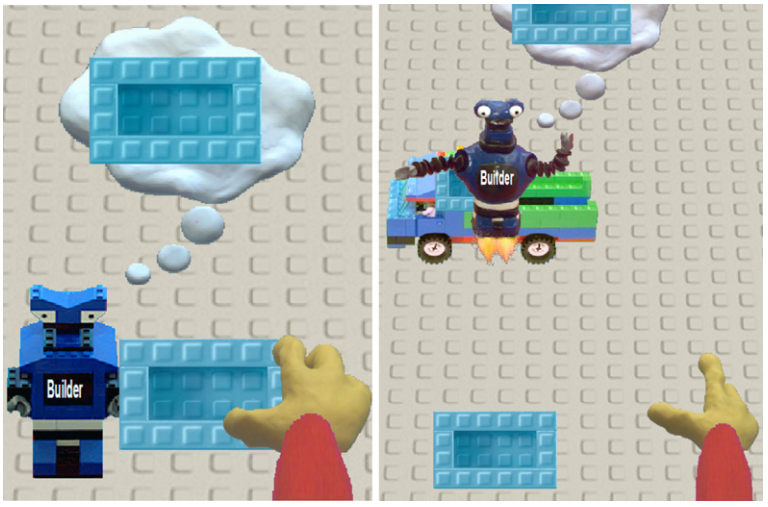


Fig. 11. First execution method: dropping the working box on a robot and watching it work.

Fig. 12. Two robots connected through a bird–nest pair.

Repeating the flip operation returns the programmable object to its original state, and the notebook of controls returns to its back.

Table 6 presents an assortment of sample controls and sensors, selected for the purpose of presenting the varieties of this kind of elements. It should be

Please cite this article as: L. Morgado, K. Kahn, Towards a specification of the ToonTalk language, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.10.002

Fig. 13. Dispatching one robot, with the width of a small picture of a doll, and 6 s later.

Fig. 14. Sending one number to the robot on the back of the picture of a doll, using the bird.

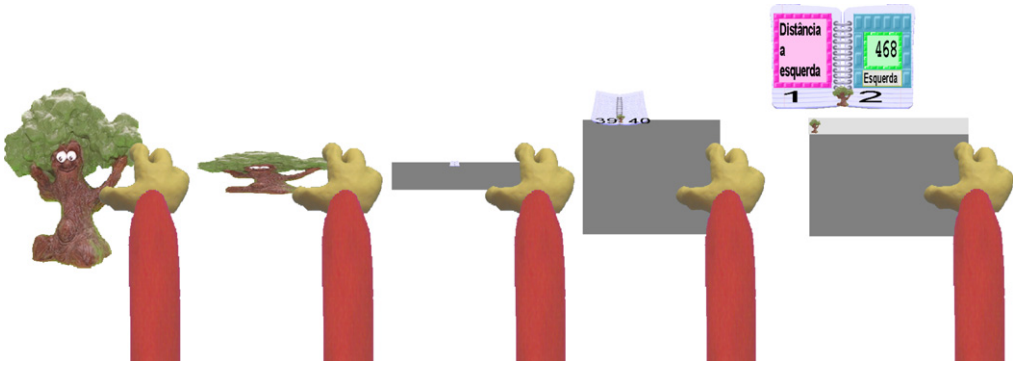


Fig. 15. Flipping a programmable object to access its contents and notebook of controls.

noted that sensors and controls commonly present some animated cue, in order to render them distinct from regular, static-looking objects. Pads have a "light marquee" outline, and picture controls flash regularly, for example.

10. Communication semantics: "bird/nest algebra"

In ToonTalk, inter-process communication is performed by means of message passing, based on two kinds of object primitives: birds and nests (*vd.* Table 2). Their use has been mentioned above regarding methods of execution or debugging, but there are complexities that warrant a more detailed description.

The basic idea is this: a bird is associated with its nest, and will deliver into that nest the object primitives that are dropped on it—this constitutes a *bird-nest communication channel*. That nest will store the received object primitives as a first-in-first-out list (*vd.* Table 4).

The complexities arise from a few properties of birds and nests when combined with other ToonTalk primitives:

1. the "magic wand copy" primitive (*vd.* Table 2) can be applied to birds and/or nests, resulting in a duplication of either or both ends of a bird–nest communication channel;
2. two nests can be combined, in effect "merging" the receiving end of two different communication channels (*vd.* Table 4);
3. it is possible to place contents directly into a nest, without using any bird at all (*vd.* Table 4), in effect "pre-loading" data into the receiving end of the channel;
4. not all object primitives can be dropped on a bird,[4] to be sent: some can only be sent if they are part of an array (i.e., inside a cubby box).

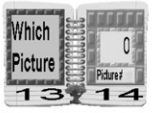
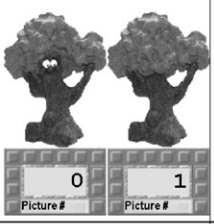
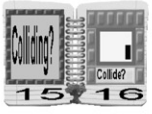
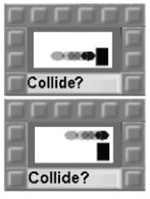
[4]There is no fundamental reason for this: it is a user-centered design option, resulting from observing confusion among early beta testers when birds or nests were dropped on birds.

Please cite this article as: L. Morgado, K. Kahn, Towards a specification of the ToonTalk language, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.10.002

We will illustrate the working of this method of communication under these complexities. For the sake of clarity, different birds and/or nests will be labeled differently, but as mentioned previously, the labels themselves have no meaning code-wise.

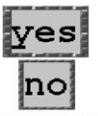
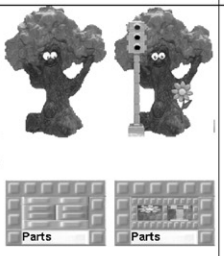
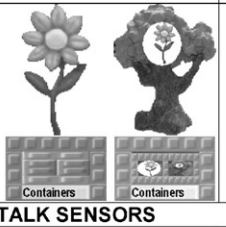
The first situation is *bird duplication*, starting from a simple bird–nest pair: $Bird_A$ and $Nest_A$. By copying the bird (with the magic wand, for instance), we duplicate the sending end. Now, object primitives dropped on either copy of $Bird_A$

Table 6
ToonTalk controls and sensors

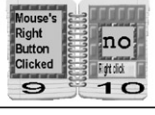
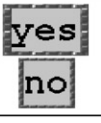
CONTROLS OF PROGRAMMABLE OBJECTS			
Image and name	**Description**	**Readable / Writable**	**Possible values**
	Distance from the left side of the containing image or viewport, measured from the center. Writing to it (dropping a number pad on top or editing with the keyboard) changes the horizontal position of the picture. Another control with a similar purpose: **From Bottom**.	R + W	Rational numbers.
	The current horizontal speed, where 0 means stopped, and a negative value means that the movement is from right to left. Another control with a similar purpose: **Speed to Top**.	R + W	Rational numbers.
	Width of the programmable object, relative to the containing image or viewport. Another control with a similar purpose: **Height**.	R + W	Positive rational numbers.
	Index of the image providing the current looks of the picture (a ToonTalk picture can be a sequence of images). For synthetic pictures, and text and number pads, it controls the foreground color. 	R + W	Positive integers, from 0 to the number of internal images.
	Shows with animation whether a programmable object is "colliding" with another ToonTalk element (*i.e.*, if some part of it occupies the same coordinates as another element) – including environment tools. Can be changed to make them "uncollide", which causes the programmable object to move slightly aside, to end the collision. Other controls with similar purposes: **Left Right Hit?** (only detects collisions primarily due to horizontal motion), and **Up Down Hit?** (only detects collisions primarily due to vertical motion).	R + W	
	If a programmable object is colliding with another, this control acts like a surveillance camera showing that other programmable object. Dusty can be used to vacuum the control contents and spit a "Looks" control (described below), so that a program can access that other object's contents and controls.	R	Any picture. 
	Determines whether a programmable object is fully visible, invisible or partly transparent. The programmer can change the current value by pointing to or holding the sensor and pressing the "+" and "-" keys.	R + W	
	Some programmable objects may have embedded animations. This shows whether those embedded animations are still running (Animation Finished is "no"), or have run to its full extent (Animation Finished is "yes"). If the embedded animation never ends (the frame list is cycling), the value is always "no": it is not "yes" at the end of an animation cycle.	R	

Please cite this article as: L. Morgado, K. Kahn, Towards a specification of the ToonTalk language, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.10.002

Table 6 (*continued*)

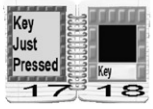
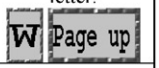
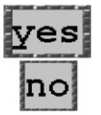
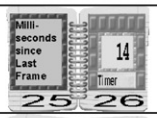
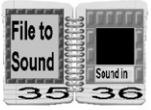
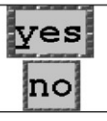
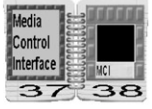
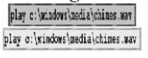
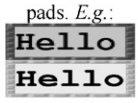
	Shows whether a programmable object is being held by the hand of the programmer's avatar or not. Another control with a similar purpose: **Just Dropped** (turns "yes" momentarily when a programmer's avatar drops the object).	R	
	Shows whether a programmable object is being pointed at by the hand of the programmer's avatar (*i.e.*, under the focus of the cursor) or not.	R	
	Provides access to the appearance of the programmable object (typically, a picture or pad), which can be edited as any picture/pad (erased, things can be dropped on it, etc.). It can also be flipped to provide access to the contents of the back.	R + W	Front of picture  Back of picture 
	Shows which programmable objects are part of this one. Only pictures may have parts on the front, but all programmable objects may have parts on their back. The sensor can be flipped to make it display the parts on the back. Dusty can be used to access these subpictures and manipulate their contents or controls. 	R	Array of programmable objects: pictures, number pads, text pads.
	Shows which programmable object this picture is either part of, or contained in the back. Dusty can be used to access these encompassing programmable objects and manipulate their contents or controls. 	R	Array of programmable objects: pictures, number pads, text pads.

TOONTALK SENSORS

Image and name	Description	Readable /Writable	Possible values
	Displays the current horizontal speed of the computer system's mouse. Another sensor with a similar purpose: **Mouse's Up Speed**. 	R	Integer numbers.
	Indicates whether the right button of the computer system's mouse was just clicked or not. Other sensors with similar purposes: **Mouse's Left Button Clicked**, **Mouse's Middle Button Clicked**, **Mouse's Left**	R	

Please cite this article as: L. Morgado, K. Kahn, Towards a specification of the ToonTalk language, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.10.002

Table 6 (*continued*)

	Button Down, **Mouse's Middle Button Down**, **Mouse's Right Button Down**.		
	Momentarily displays the value of the key that was pressed on the keyboard. Before and after that moment, it has no value.	R	A text pad indicating the letter. 
	Indicates the key that was most recently pressed on the computer system's keyboard.	R	A text pad indicating the letter. 
	Indicates whether one of the shift keys is currently being pressed on the keyboard or not. Another sensor with a similar purpose: **Control Button Down**. (The current version of ToonTalk does not have an "Alt Button Down" sensor.)	R	
	Indicates how many milliseconds have elapsed since the last animation frame.	R	Integer numbers.
	The value of this sensor is constantly changing, in pseudo-random manner.	R	Any integer number between 0 and 999.
	Presents, as a picture, the current contents of the MS Windows system clipboard. Dusty can be used to vacuum it and spit, in order to access these contents. ToonTalk elements can also be dropped on it, to copy them to the clipboard.	R + W	Any ToonTalk object primitive (contents from other programs are converted).
	The programmer can drop a filename or URL on this sensor, which will respond by producing a sound pad with the sound found in the matching location. Another sensor with a similar purpose: **File to Picture**.	R + W	Any sound pad.
	Determines whether the current representation of the ToonTalk programmer's avatar (*i.e.* hand, helicopter) is visible on-screen or not.	R + W	
	The programmer can drop on this sensor a text pad with a command for Microsoft Windows Media Control Interface. The sensor responds with the resulting sound pads. *E.g.*, dropping the text pad "play c:\windows\media\chimes.wav" results in a sound pad for the sound in that file.	R + W	Commands and their results. *E.g.*: 
	The programmer can drop on this sensor a text pad for the default Text-to-Speech engine installed in Microsoft Windows. The sensor responds with a sound pad capable of playing synthesized speech for the original text. *E.g.*, dropping the text pad "Hello" results in a sound pad for the word "Hello".	R + W	Text pads and resulting sound pads. *E.g.*: 

Please cite this article as: L. Morgado, K. Kahn, Towards a specification of the ToonTalk language, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.10.002

Table 6 (*continued*)

	Allows checking and editing the appearance of the current house façade. It can be read to be used in comparisons, but it cannot be retrieved as a picture for general use. Other sensors with similar purposes: **Wall Decoration**, **Roof Decoration**.	R+W	Any picture.
	This pair of sensors presents to the programmer the city block where the current house is located (avenue and street – there are four lots per block, but no way to check which one is the current). They can be edited by dropping number pads on them or with the '+' and '-' keys, for dropping on a departing truck, in order to determine where that truck's new house should be built.	R+W	Read: current status as a text pad only for comparisons. Write: any integer.
	Displays the currently active language for the ToonTalk environment. This affects the help text and the command letters for operating the tools, but can be checked by programs to impact their behavior.	R	Text pad with name of language (current options: English, Portuguese, Swedish).
	This bird establishes a link with foreign code. By giving it a box with a DLL name and a bird, it will fly out and give that "regular" bird another one, which can then be used to invoke functions in the external DLL.	W	Box with two holes: in the first is a pad with the name of a DLL file and in the second is a bird.

Fig. 16. Duplication at one end of a communication channel.

will be delivered into the same nest, $Nest_A$ as shown on the left half in Fig. 16.

A second situation is that of *nest duplication*, starting from a simple bird–nest pair: $Bird_B$ and $Nest_B$. By copying the nest (with the magic wand, for instance), we duplicate the receiving end, as shown on the right half in Fig. 16. Now object primitives dropped on $Bird_B$ are duplicated (copied), and each $Nest_B$ receives a copy. A different perspective on this could be that the object primitive is turned into a template upon being dropped on a bird, and from that template several instances are created, one for each nest.

All further complications of bird or nest duplication follow this same rule. For instance, once we have $Bird_{A1}$ and $Bird_{A2}$, we could duplicate $Nest_A$ and get $Nest_{A1}$ and $Nest_{A2}$. And both birds would deliver copies of messages to both nests.

In these situations, issues arise of synchronization of message delivery. For instance, if we assume that there are two copies of $Bird_A$, which we will refer to as $Bird_{A1}$ and $Bird_{A2}$, in what order do they place messages into the queue at $Nest_A$? Conversely, if there are two copies of $Nest_B$, which we will refer to as $Nest_{B1}$ and $Nest_{B2}$, in what order are messages duplicated and delivered into their queues?

Here, ToonTalk considers that delivery is independent of speed and asynchronous. That is, items are guaranteed to arrive into a nest in the same order as they were sent via the same bird, but not necessarily in an uninterrupted sequence.

For instance, if $Bird_{A1}$ is given the sequence {1,2,3,4,5} and $Bird_{A2}$ is given the sequence {A,B,C,D,E}, then $Nest_A$ may not only receive {1,A,2,B,3,C,4,D,5,E}, but also {1,2,A,3,B,C,4,D,5,E}, or even {A,B,C,D,E,1,2,3,4,5}. Similarly, if $Bird_B$ is given the sequence {1,2,3,4,5}, $Nest_{B1}$ and $Nest_{B2}$ will each receive a copy of this sequence in the same order, but the timing of arrival of the elements may vary: a "1" might arrive at $Nest_{B1}$ just after another "1" arrived at $Nest_{B2}$, or the other way around, or even all elements may be delivered into a nest before any are delivered into the other nest at all.

A third situation is that of *nest merging*. As shown in Table 4, a nest can be dropped on another, and this causes both nests to be merged, in effect merging the receiving ends of both communication channels. For instance, if $Nest_A$ and $Nest_B$ are merged into $Nest_{AB}$, then both $Bird_A$ and $Bird_B$ will deliver messages into $Nest_{AB}$, as will other copies of these birds. However, the merged nest cannot be unmerged: any operation (i.e., duplication) will operate on the merged nest, $Nest_{AB}$.

However, $Nest_A$ and $Nest_B$ might have previously existing copies, i.e., $Nest_{A1}$, $Nest_{A2}$, $Nest_{B1}$, $Nest_{B2}$, … If two nests are merged, creating for instance $Nest_{A1B1}$, this does not affect the other copies. This way, any $Bird_A$ will deliver copies of messages into $Nest_{A2}$ and $Nest_{A1B1}$, but not on $Nest_{B2}$: and any $Bird_B$ will deliver copies of messages into $Nest_{B2}$ and $Nest_{A1B1}$, but not on $Nest_{A2}$.

A fourth situation is that of *direct nest queue manipulation*. As listed in Table 4, an object primitive can be placed into an empty nest, in effect pre-loading data into the nest queue. However, we must clarify that this can only be done exactly once: when the nest is empty: it is not possible to preload two or more object primitives into the nest, since the moment it has any contents all attempts to directly drop more elements on top of it will result in the combination of the topmost element with the dropped-on one. Therefore, only birds can cause a nest to queue-up messages. However, one can picture a process holding the nest that attempts to empty it and place a new element inside, before further messages arrive, in effect "inserting" extra elements in the message arrival sequence, if not in the queue itself.

A final situation is that of *valid object primitives for messages*. In current implementations of ToonTalk, the following object primitives cannot be sent in messages, unless they are inside a cubby box (i.e., part of an array): birds, nests, bombs, trucks, robots, and notepads. One could consider this limitation somewhat arbitrary (as mentioned in footnote 3, it is merely a user-centered design option) and therefore unnecessary, but for one situation that warrants a close analysis: nests. Allowing nests to be sent as a message themselves would lead to situations where a nest would be placed on top of another. One may speculate that there could be two different interpretations as to the result of this. One might see this as meaning that the nest sent via bird would be placed on top of the receiving nest. This bottom nest (and its queue) would then for all purposes be inaccessible to the receiving process, unless it regularly checked for this situation of "nest on top of nest". This, we believe, is highly undesirable, since it would potentially lead to many situations where systems resources would be wasted in these "hidden queues". Therefore, we propose that a formal ToonTalk specification considers that upon arrival of messages at the end of a channel, these are being "dropped" on top of the nest. Consequently, a nest being sent via bird would be a "nest dropped on nest" situation, resulting in the merging of both nests—a situation that would be semantically identical to a bird-duplication on the part of the sending process.

11. Final thoughts

The ToonTalk software product is the proof-of-concept of animated programming (or action-based programming, if seen from the perspective of the programmer's actions, not their representation). Furthermore, it is also a proof-of-concept of the possibility of conducting powerful programming activities within a virtual world, entirely in terms of visual manipulation of elements within that world. However, the ToonTalk language itself lacks an actual specification, and as a result previous works have not rendered clear the distinction between the ToonTalk programming environment and the ToonTalk programming language.

Our contribution here lacks the formal approach of a typical language specification. Determining

Please cite this article as: L. Morgado, K. Kahn, Towards a specification of the ToonTalk language, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.10.002

how to perform such a formal specification for an animated programming language is a very challenging problem, and would in itself constitute an interesting research project. A complete specification could also identify a language kernel, capable of creating an equivalent full-fledged language specification within the language itself. What we do provide is a contribution towards such a specification, by clarifying the distinction between the environment and the language, describing the building blocks of this language, and the meaning of their combinations.

Acknowledgments

Many people have contributed with comments, suggestions, reflections, and bug reports to the development and refinement of the ToonTalk language in the form of its implementations as the commercial products ToonTalk, ToonTalk 2, and ToonTalk 3. We would like to thank them all. Also, we would like to thank members of the ToonTalk mailing list for theirs comments and insights, particularly Gordon Simpson, Jack Waugh, Jakob Tholander, Lennart Mogren, Mikael Kindborg, Richard Noss, Tiago Correia, Yishay Mor, and Ylva Fernaeus.

References

[1] K. Kahn, ToonTalk™—an animated programming environment for children, in: D. Harris (Ed.), & R. Bailey (assistant Ed.), NECC '95 Proceedings, 17–19 June 1995, Baltimore, MD, Towson University, Towson, MD, USA, 1995.

[2] K. Kahn, ToonTalk™—an animated programming environment for children, Journal of Visual Languages & Computing 7 (2) (1996) 197–217.

[3] Animated Programs, ToonTalk—Making Programming Child's Play, 〈http://www.toontalk.com/〉 (retrieved 09.05.06).

[4] Playground Project, 3rd Annual Report—September 2001, 〈www.ioe.ac.uk/playground/RESEARCH/reports/finalreport/〉 (retrieved 09.05.06).

[5] WebLabs Project, Introduction, 〈http://www.lkl.ac.uk/kscope/weblabs/〉 (retrieved 29.11.06).

[6] L. Morgado, M.G.B. Cruz, K. Kahn, Using ToonTalk in kindergartens, in: Proceedings of the IADIS International Conference e-Society 2003, vol. II, IADIS, Lisbon, Portugal, 2003, pp. 988–994.

[7] L. Morgado, M.G.B. Cruz, K. Kahn, ToonTalk in kindergartens: field notes, Journal of Digital Contents 1 (1) (2003) 111.

[8] L. Morgado, Framework for computer programming in preschool and kindergarten, Doctoral Thesis, Universidade de Trás-os-Montes e Alto Douro, Vila Real, Portugal, 2006.

[9] T.S. McNerney, From turtles to Tangible Programming Bricks: explorations in physical language design, Personal and Ubiquitous Computing 8 (5) (2004) 326–337.

[10] P. Frei, V. Su, B. Mikhak, H. Ishii, Curlybot: designing a new class of computational toys, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ACM Press, New York, NY, 2000, pp. 129–136.

[11] H.S. Raffle, Topobo: a 3-D constructive assembly system with kinetic memory, Master's Dissertation, School of Architecture and Planning, Massachusetts Institute of Technology, Cambridge, MA, 2004.

[12] T. Green, A. Blackwell, Cognitive dimensions of information artefacts: a tutorial, version 1.2, 〈http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDtutorial.pdf〉,1998 (retrieved 26.06.07).

[13] D. Edge, A. Blackwell, Correlates of the cognitive dimensions for tangible user interface, Journal of Visual Languages and Computing 17 (2006) 366–394.

[14] K.N. Whitley, Visual programming languages and the empirical evidence for and against, Journal of Visual Languages and Computing 8 (1997) 109–142.

[15] G. Dowling, A. Tickle, K. Stark, J. Rowe, M. Godat, Animation of complex data communications concepts may not always yield improved learning outcomes, in: Proceedings of the 7th Australasian Conference on Computing Education, Australian Computer Society, Darlinghurst, Australia, 2005, pp. 151–154.

[16] R.E. Mayer, M. Hegarty, S. Mayer, J. Campbell, When static media promote active learning: annotated illustrations versus narrated animations in multimedia instruction, Journal of Experimental Psychology: Applied 11 (4) (2005) 256–265.

[17] B.B. Bederson, A. Boltman, Does animation help users build mental maps of spatial information? in: Proceedings of the 1999 IEEE Symposium on Information Visualization, IEEE Computer Society, Washington, DC, 1998, pp. 28–35.

[18] W. Schnotz, T. Rasch, Enabling, facilitating, and inhibiting effects of animations in multimedia learning: why reduction of cognitive load can have negative results on learning, Educational Technology Research and Development 53 (3) (2005) 47–58.

[19] A.F. Blackwell, Pictorial representation and metaphor in visual language design, Journal of Visual Languages and Computing 12 (2001) 223–252.

[20] M. Kindborg, Concurrent comics—programming of social agents by children, Linköping studies in science and technology, Dissertation No. 821, Department of Computer and Information Science, Linköpings universitet, Linköping, Sweden, 2003.

[21] J.-C. Mézières, P. Christin, Ambassador of the shadows, (L. Mitchell, Trans.), Hodder & Stoughton, London, UK, 1984. (Valerian: Spatio-Temporal Agent, ISBN 0-450-05767-4).

[22] K. Kahn, Generalizing by removing detail: how any program can be created by working with examples, in: H. Lieberman (Ed.), Your Wish Is My Command: Programming By Example, Morgan Kaufmann, San Francisco, CA, 2001 (Chapter 2).

Please cite this article as: L. Morgado, K. Kahn, Towards a specification of the ToonTalk language, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.10.002