Comic Strip Programs: Beyond Graphical Rewrite Rules

Mikael Kindborg and Kevin McGee
Department of Computer and Information Science
Linköping University, S-58183 Linköping, Sweden
{mikki, kevmc}@ida.liu.se

Abstract

Comics and programs both are static representations of something dynamic. While a comic book almost looks and feels like an animated cartoon, the source code of a program seldom resembles the visible runtime behaviour. By using comic strips to represent concurrent events, graphical programs that feature interactive, animated characters and objects can be expressed in a visually direct way, making it easier for both children and adults to create their own computer programs. Compared to graphical rewrite rules, comic strip programs have a potential for increased expressiveness and flexibility, because of how the semiotics of comics can be used to express a wide variety of situations. There are also challenges with using comics as a program representation, for example, comics have no signs for conditionals, concurrency, and generalisations.

1. Introduction

One approach for making programming easier, is to use a program representation that looks similar to and directly maps to the runtime representation [1]. If the source code of a program directly maps to and has a strong similarity to that which is seen on the display when running the program, the gap between the two representations can be bridged, and the need for difficult and error-prone mental transformations could be reduced, thus making programming easier.

A visual representation that is interesting in this respect is comics. Like a program, a comic is a static representation of something dynamic. The medium of comics gives a very direct impression of the action going on in the story. To the comic book reader, the characters in a comic almost look like they are moving and they almost sound as if they are speaking. For programs that consist of interactive graphical objects, like games, graphical simulations, and interactive pictorial stories, the signs used in comics have the potential to describe the behaviour of the objects in a way that strongly resembles the visual result of running the program.

In comics, panels are used as the basic temporal device for communicating dynamics in a static medium [2], [3]. A static representation is straightforward to edit, which is essential to programming, and provides an overview that is independent of the real-time flow of a dynamic representation. Inside panels, comic book artists use a rich vocabulary of contextual signs ("markers") for representing dynamic and abstract features. Examples include motion markers (e.g. speed lines and ghost images), voice balloons, and sound words (onomatopoetic symbols). Importantly, such markers are shown in the immediate visual context of the character or object having the feature represented by the sign. This presentation technique takes advantage of the way human perception can quickly perceive a situation "at a glance", and creates a high degree of visual directness. Additionally, many people are familiar with comics, and even though there are cultural differences, for example between Western European and Japanese comics [3], the sign language of comics is well-known within the comic book reading community. This could mean that people who want to learn to program would feel familiar with the signs used in a comic strip programming language.

One way to use comics for programming of graphical objects, is to have conditional comic strips that represent events in a program [4]-[7]. Such comic strip programs share many characteristics with graphical rewrite rules [1], [8]-[16], also called visual before-after rules. The before-part of a rule consists of a picture representing a part of the world. When the before-part matches the current world state, the world is changed to the state shown by the picture in the after-part of the rule. Such rules look very similar to what actually happens when the program executes. Graphical rewrite rule systems commonly use a grid-based world model, which makes it straightforward to specify the before and after parts of a rule in a spatially clear and non-ambiguous way.

Graphical rewrite rules are an example of an analogical representation, a representation that has a structural similarity to what it represents [1]. In semiotics, such representations are called iconic signs. An icon is a sign that has a similarity or likeness to the object it signifies; it imitates the signified. A symbolic sign, by contrast, is arbitrary and gets its meaning by convention [17].

Analogical representations and iconic signs are usually easy to understand, but not everything can be represented this way. In programming, there are constructs that have no iconic representation. Comics offer a way to integrate symbolic signs into the context of iconic signs, which can help understanding symbolic representations.

This paper presents a way to use comics for event-based visual programming, and discusses how the signs of comics can be applied to programming. Finally, it is suggested that comic strip programs have the potential to go beyond graphical rewrite rules in terms of expressiveness and flexibility, while maintaining a strong visual directness between the program and the runtime display.

2. Comic strip programming

Several prototypes have been developed to test the idea of comic strip programming. Both low-fidelity paper prototypes and very high-fidelity computer prototypes have been studied [4]-[7]. The domain focused by these prototypes have been simple games and interactive stories with graphical objects. In the following, examples of comic strip programs will be given using screenshots from the most recent system being developed, called ComiKit.

In comics, the action sequences that make up the story are communicated using strips of panels, where each panel shows a part of the action. If we introduce the notion of conditional strips, comic strips can be used to describe events in a program. In a conditional comic strip, the first panel is the precondition for the actions in the subsequent panels. Such strips describe non-linear and potentially concurrent events, rather than a sequential flow of actions in a story.

A comic strip program consists of graphical objects that typically depict figurative characters and items. An object can have any graphical representation, though, as it is the programmer who draws or chooses the pictures. Each object can have one or more pictures. For example, a character could have two pictures, one for happy mood and one for sad mood. The pictures represent different states of an object.

Each object may have one or more comic strip events associated with it. The program is run by placing characters on a play area and moving them with the mouse, causing events to trigger. Events are continuously monitored and executed by the runtime engine (the interpreter).

Figure 1 shows an example of event strips created by two girls in fifth grade, during a field study in a school. Figure 2 shows what it looks like when the game is played. The preconditions used in Figure 1 are picture-matching and touching another object. The action used is picture-changing. Picture matching works by checking if an object has the picture shown in the precondition. If, for example, the character called "John" in Figure 1, would be on the play area having a picture of him sleeping, the coughing character would not infect him, because the first event in Figure 1 would not match.

The situation shown in the precondition of an event strip can be thought of as an example of a situation that

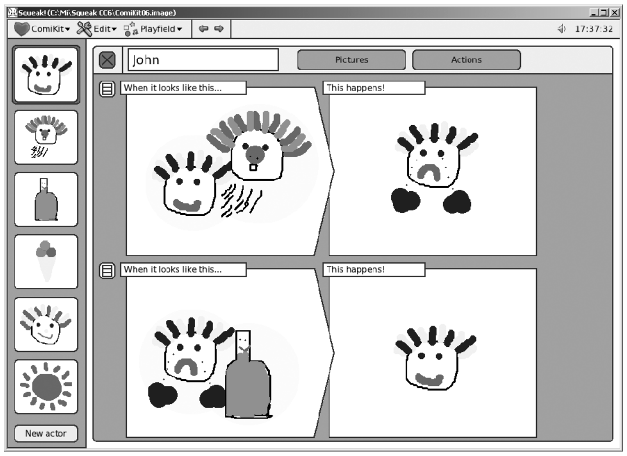
Squeak! (C:\Mi\Squeak CC6\ComiKit06.image)
ComiKit
Edit
Playfield
17:37:32
John
Pictures
Actions
When it looks like this...
This happens!
When it looks like this...
This happens!
New actor

Figure 1. Programming with event strips. This example shows two events for a character called "John". In the first event strip John becomes infected and gets sick. The precondition panel shows John touching another character, who has got a cough. In the action panel, the picture of John changes to show him sad and crying (the shapes below John are pools of tears). In the second event strip, he gets cured by taking medicine. The precondition panel shows the sick John touching a bottle of medicine, and the action panel changes his picture to happy.

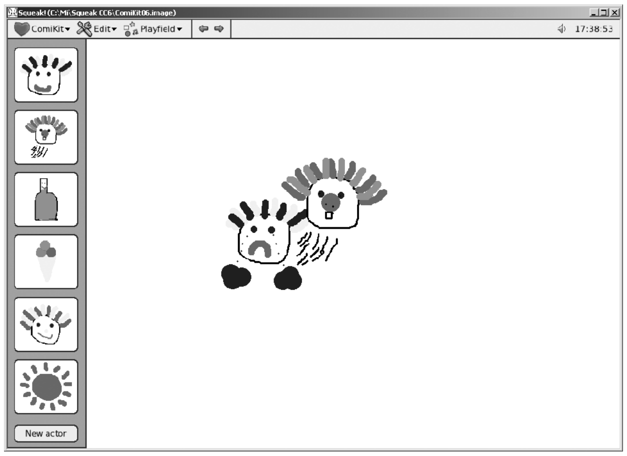

Figure 2. The play area. The player drags characters from the gallery at the left and drops them on the play area. The player can move the characters on the play area with the mouse, causing events to trigger as the characters touch each other. Here is a situation where the player has moved the coughing character with the mouse, to make it touch John. This caused the first event in Figure 1 to trigger, which changed John's picture from happy to sick.

triggers the event. On the play area, the characters need not touch each other in exactly the same way as in the precondition, for the event to trigger. Touching in any direction (left, right, above, or below), will match the precondition.

The object model that is used is similar to classes in object-oriented programming. In an event there is always a main object (similar to the notion of "self" in object-oriented programming languages such as Smalltalk). The main object is the object for which an event is defined (events belong to objects like methods belong to classes in object-oriented programming).

Dragging a character from the gallery to the play area creates an instance of that object type. Several copies (instances) of a character can be created this way (new instances can also be created in events). If there would be many copies of the character called John on the play area in Figure 2, they could all be infected by touching them with the coughing character.

The preconditions currently implemented are picture matching, touching another object, keyboard pressed, time interval, and random time interval. The actions implemented are changing the picture of an object, moving an object, delete an object, and create a new object instance. Many additional preconditions and actions could be imagined.

Several field studies of previous prototypes of the ComiKit system were made in a school together with children in grade four and five [6], [7]. The children could learn the fundamentals of comic strip programming to create programs like interactive picture-stories in one to two hours. However, basic parts of the programming model required explanation to be understood in the intended way. The main finding was that the children's intuitive interpretation of comic strip programs was that of a linear action sequence. The children had to learn programming constructs such as conditionals and concurrency to be able to successfully create programs, notions that contradicted the familiar view of comics as sequential stories.

The prototypes studied used standard rectangular panels, and the arrow-shaped precondition panel used in the current system is an attempt to make the conditional panel "look special". How this shape is understood by users has not yet been studied, however. Another problem was to put events on the "right" character. For some events it is important that the main object is properly chosen, and the children were not always aware of this.

In the prototypes used, each event could have only one action panel (this is also the case in the current system). This restriction was made primarily to simplify the implementation. Interestingly, this forced the children to break up linear action sequences into multiple events, which resulted in that the interactive stories they created could be played in a much more open-ended style than what could be expected at first [6].

3. Comic book signs and visual programming

Several comic book signs can be applied to visual programming, but there are also many symbolic signs that are needed for programming that are missing in comics.

3.1. Applying comic signs to programming

The following is a discussion of signs in comics that can be applied directly to visual event-based programs.

Characters. The appearance of a character in the program directly maps to the appearance in the runtime representation.

Panels. The panels in a comic divide time into smaller units. In a program, panels can be used to represent action sequences, and by introducing conditional panels, events with preconditions can be expressed. Note that a panel in a comic is not just a frozen moment of time. Unlike a photograph, a panel can show a situation that is extended in time. Speaking in a voice balloon, for example, is not an instantaneous action. Objects in a program could also perform multiple activities within the timespan of a panel.

Meetings. Characters in a panel are commonly shown meeting, simply by positioning them close to each other. This can be used in a program to show a collides (touches) condition in a visually direct way.

Motion markers. Comic books feature signs like speed lines and ghost images that can be used to show the motion of objects in a visual program. An example of a ghost shadow motion marker is given i Figure 3.

Transitions. In comics, as in film, cuts between perspectives and scenes are common. For example, the panels in an event strip could show objects at different places, as in Figures 3 and 4.

Inner panels. An inner panel is a panel inside another panel. Such panels are used to show something that happens at the same time, but at another place. These panels can be used by event strips to make location-independent references to objects (similar to global variable references). See Figure 5 for an example.

Voice balloons and text boxes. Balloons can be used to output text that is "spoken" by the characters in the program. Contextual text boxes could be used to represent symbolic programming constructs that are awkward or impossible to express pictorially, for example, textual and numerical object properties.

3.2 Signs that are missing in comics

The following are examples of programming constructs for which there are no corresponding signs in comics.

Conditionals. Strips in a comic book does not have conditional panels. Introducing a special sign, like an arrow between the first and the second panel, or an arrow-shaped precondition panel, could help to communicate the special status of the first panel.

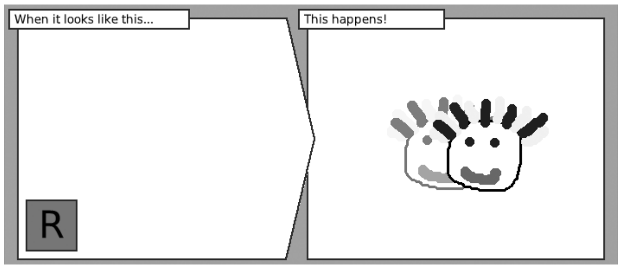

Figure 3. Motion marker. This example shows an example of a ghost image motion marker. The precondition contains a picture of a keyboard key. When pressing the R key, the character moves to the right.

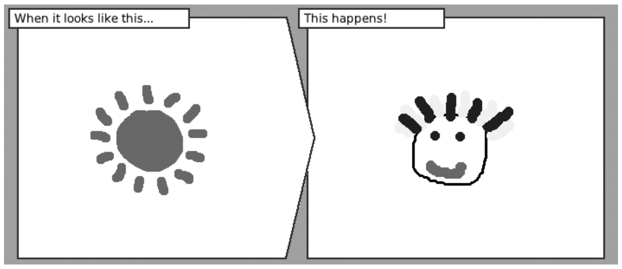

Figure 4. Panel transition. In this strip the panels show different parts of the world. When there is a sun shining in the world, the main character will become happy, regardless of its current look. This strip is an example of what is called an aspect-to-aspect transition in comics.

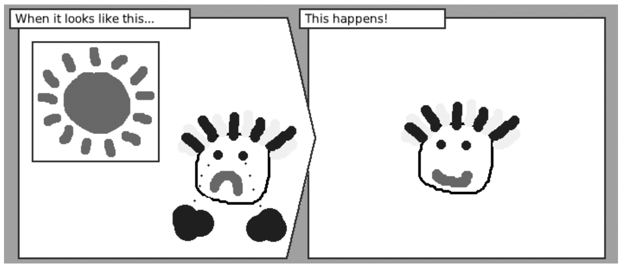

Figure 5. Inner panel. Since the Sun is inside an inner panel, it can be located anywhere for the event to trigger, but it will trigger only when the main character looks sad.

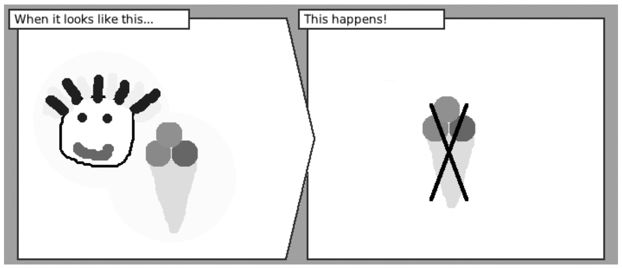

Figure 6. Delete action. In this event the main character "eats" an ice cream when touching it. The cross signifies the delete action, and is an example a contextual marker.

Non-sequential control structures and concurrency. Events in a comic strip program can be triggered in any order and execute concurrently, depending on, for instance, how the player interacts with a game. This stands in contrast to the reading order of comics, where strips represent a sequence of actions.

Deletion of objects. Comics use situations in the context of the story to show when something disappears. There does not seem to exist generic signs for deleting objects. The X-shaped cross in Figure 6 is an attempt at creating a generic sign for the delete action.

Generalisation. An example of generalisation that is handy in a precondition panel is being able to say that an object should match objects of the same kind regardless of the picture (picture generalisation), or that it should match any existing object (type generalisation). Such generalisations can reduce the number of strips needed to express an event [7]. Comics depict concrete situations and have not developed signs for these kinds of generalisations. Possible signs could be a circle with a question mark for matching any object, and an outline of the first picture of an object for matching any object picture. Note that we get picture generalisation "for free" when the main object is not present in the precondition panel (see Figures 3 and 4). However, this technique can not be used when touching other objects, because then the main object must be present in the first panel.

Variables. In a way, an object instance can be thought of as a global variable with a value range given by the pictures defined for the object type (like an enumerated data type). Objects can check the state of another object by including that object in the precondition (without touching it). However, if there are several instances of the other object, there is currently no way to identify them. Global names could be used for this, or an object could be linked to another, creating something similar to an instance variable referring to another object.

Negation. Expressing negation pictorially is another example of something that comics have no generic signs for. How would one express that an object is not touching another object, or that an object should not have a particular picture? One solution could be to negate the entire precondition panel, or to have two panels following the precondition, labelled "YES" and "NO", which would be similar to an if-else clause.

4. Beyond graphical rewrite rules

The idea of representing programs as picture sequences has also been used by systems based on graphical rewrite rules (also called before-after rules). Examples of such systems are BITPICT [8], Cartoonist [12], KidSim [13], [14], Stagecast Creator [1], and AgentSheets [11], [15], [16]. This section discusses limitations of graphical rewrite rules and how these limitations could be addressed by the visual language of comics.

4.1. Limitations of graphical rewrite rules

Graphical rewrites. Typically, graphical rewrite rules define before-after rewrites of the world state. This makes it problematic to refer to objects in a flexible way, or to objects that could be located anywhere in the world. Most systems that use graphical rewrite rules seem to have the restriction that the before part must be the same size and show the same location of the world as the after part, and except for objects created or deleted, the same objects must be present in both parts.

Iconic signs. Graphical rewrite rules consist of pictures of domain objects. This makes the representation iconic to the runtime representation, but places limits on what can be expressed. For example, in Stagecast Creator [1], symbolic program constructs are placed outside of the pictorial part of the rule.

The grid. Most graphical rewrite rule systems use a grid-based world model. There are several limitations of using a grid; objects must have fixed sizes, must be placed at discrete locations, can not overlap, and object motion is restricted and jagged. Interacting with objects that are confined to a grid in a flowing, direct manipulation style, e.g. continuously dragging an object with the mouse, is impossible. It should be noted that it is possible to use before-after rules with other models than a grid. One example is ChemTrains [9], [10], a system which uses a topological model where any topology based on containment and connected locations can be used, including a grid.

4.2. Using the visual language of comics

In the following it is discussed how the visual language of comics could be used to address some of the problems discussed above.

Panel transitions. McCloud identifies six types of panel transitions that are used in comics [3], pp. 70-72: Moment-to-moment (e.g. open eye – closed eye), action-to-action (e.g. glass being poured – someone drinking), subject-to-subject (e.g. two people talking – close up of a ringing phone), scene-to-scene (e.g. at home – at a football game), aspect-to-aspect (several aspects or moods of something, e.g. sun shining – child playing), and non-sequitur (has no obvious meaning). The transitions that are used by graphical rewrite rules are similar to moment-to-moment and action-to-action transitions where the panels show the exact same perspective. By adopting additional kinds of transitions, making before-after pictures less tightly coupled, the restrictions imposed by before-after rewrites could be relaxed. Visual programming languages could take advantage of increased expressiveness and flexibility, while preserving a direct visual mapping between the program and the runtime result. In addition to various transitions, different perspectives could be used. The precondition could for example contain a close-up of a control panel with a button in a pressed state, and the action could contain a full view of a rocket being launched.

Contextual signs. Comics mix iconic and symbolic representations, and use a wide variety of contextual signs to visualise what is going on inside the panels. Symbols, like motion markers, are used to visualise dynamic aspects within a static picture. Such markers are shown in the context of the object being modified by the sign, which creates a direct mapping between the symbolic sign and the iconic object. The tight integration of signs brings the picture to life, transforming the static image to a dynamic one in the mind of the reader. Marker signs also extend the time span of a panel, making it possible to visualise several actions within a panel.

Visual programming languages could use contextual signs to represent symbolic attributes of objects in a visually direct way. Textual and symbolic signs, like numbers and mathematical operators, could be integrated into the context of the graphical objects in a program. The way comics integrate text and pictures creates a quality of directness; the representation becomes a whole which guides the reader by providing mappings within the representation itself.

AgentSheets is a system that takes advantage of mixing textual and iconic signs [15]. However, several programming constructs in AgentSheets are text-only and rules tend to look more like illustrated texts than like pictures with text [15], [16]. The similarity between the program and the runtime result is not as strong as it potentially could be by adopting comic book techniques.

Gridlessness. Objects that live in a gridless world can be interacted with in a much more flowing way than what is possible when restricted to locations in a grid. To make direct manipulation "feel right", the motion must be continuous. In a gridless model, objects can be any size, they can move in any direction at any speed, and they can overlap each other. Objects in a gridless model can refer to each other independently of their relative location in a grid. Inner panels are an example of how a visual programming language can use signs from comics to refer to objects in a location-independent way. Furthermore, a continuous coordinate system makes it possible to represent Logo-style turtle geometry in a visually direct way with motion markers [7].

It should be emphasised that using a grid also has several advantages, like reduced brittleness, relational transparency, clarity of spatial relationships, implicit communication between adjacent grid-cells, and regularity [11]. A gridless system could use an editing grid as a way to get some of these advantages. It should also be possible to extend a grid-based model to become more flexible, e.g. by allowing overlapping objects. A technical advantage with using a grid-based model is execution speed, in particular when checking if an object is close to objects in adjacent grid cells.

5. Conclusion

The comic strip program examples given in this paper are intended to demonstrate that the visual language of comics has the potential to represent event-based visual programs in an expressive and flexible way that directly maps to the runtime representation.

Comics have signs for expressing action sequences in a visually direct way, but lack signs for expressing conditionals, concurrency, generalisation, and other programming constructs. New signs have to found or invented to express programming concepts in a clear way, and people who want to create comic strip programs need learn these signs to become successful programmers.

An important contribution of comics is the use of contextual symbolic signs. Such signs usually have no meaning in isolation, they are an effect of what happens to the objects in a panel. Speed lines and ghost images, for example, are used to signify that something is moving. In our imagination, a character with speed lines "looks like" it is moving, even though in the medium nothing moves at all. Another example is onomatopoetic symbols, signs that imitate sound, for example the letters **Zzzzzz**, which are used to signify that someone is sleeping. In the imagination of the reader, the sign becomes the sound; seeing the sign is almost like hearing the sound. There are many other signs that once known can be said to "look similar" to what they signify, even though they are symbolic signs. Within the sign system of comics, these contextual signs take on a very direct, sometimes almost iconic, quality.

Contextual signs could be used in visual programming to visualise how objects are effected by the operations in a program. Such signs have the potential to address some of the limitations of analogical and iconic representations when it comes to expressing symbolic operations. Signs invented by comic book artists tend to become learned and accepted by the readers as they are introduced. McCloud writes: "Within a given culture these symbols will quickly spread until everybody knows them at a glance." [3], p. 131. If the comic book style of programming would catch on, a similar evolution might take place for visual programming.

In the beginning of film art, movies were like filmed theatre. A static camera recorded actors performing on a stage. Since then, film has evolved into a genre with a rich visual language. The medium of comics has evolved in parallel with film, and comic books have transferred and transformed the language of film to fit the printed medium. Visual programming with iconic representations of domain objects also has the potential to evolve, from the static camera used by graphical rewrite rules to something that we have not yet seen.

References

[1] D.C. Smith, A. Cypher, L. Tesler, "Novice Programming Comes of Age", *CACM*, vol. 43, no. 3, pp. 75-81, Mar. 2000.

[2] W. Eisner, *Comics & Sequential Art*, Florida: Poorhouse Press, 1985.

[3] S. McCloud, *Understanding Comics*, New York: HarperCollins Publishers, 1993.

[4] M. Kindborg., A. Kollerbaur, "Graphical Tools for Description of Dynamic Models", in Proc. *INTERACT'87*, Stuttgart, Germany. 1987.

[5] M. Kindborg, "How Children Understand Concurrent Comics: Experiences from LOFI and HIFI Prototypes", in Proc. *HCC'01*. Stresa, Italy, 2001.

[6] M. Kindborg, "Comics, Programming, Children, and Narratives", in Proc. *Interaction Design and Children*, Eindhoven, The Netherlands, Aug. 28-29, 2002.

[7] M. Kindborg, "Concurrent Comics - Programming of social agents by children". Ph.D. Dissertation, Linköping University, 2003.

[8] G.W. Furnas, "New Graphical Reasoning Models for Understanding Graphical Interfaces", in Proc. *CHI'91*, New Orleans, 1991.

[9] B. Bell, "ChemTrains: A Rule-Based Visual Language for Building Graphical Simulations", Technical Report CU-CS-592-92, Department of Computer Science, University of Colorado at Boulder, 1992.

[10] B. Bell, Bringham, C. Lewis, "ChemTrains: A Language for Creating Behaving Pictures", in Proc. *IEEE Workshop on Visual Languages*, Bergen, Norway. 1993.

[11] A. Repenning, W Citrin, "Agentsheets: Applying Grid-Based Spatial Reasoning to Human-Computer Interaction", in Proc. VL'93: *IEEE Workshop on Visual Languages*, Bergen, Norway, 1993.

[12] R. Hübscher, "Composing Complex Behaviour from Simple Visual Descriptions", in Proc. *IEEE Symposium on Visual Languages*, Boulder, CO, 1996.

[13] C. Rader, C. Brand, C. Lewis, "Degrees of Comprehension: Children's Understanding of a Visual Programming Environment", in Proc. *CHI'97*, Atlanta, USA, 1997.

[14] D.C. Smith, A. Cypher, J. Sphorer, "KidSim: Programming Agents without a Programming Language", in *Software Agents*, J. M. Bradshaw, Ed. AAAI Press / The MIT Press, 1997, pp. 165-190.

[15] C Rader, G. Cherry, C. Brand, A. Repenning., C. Lewis, "Principles to Scaffold Mixed Textual and Iconic End-User Programming Languages" in Proc. *IEEE Symposium of Visual Languages*, Nova Scotia, Canada, 1998.

[16] A. Repenning, A. Ioannidou, "Agent-Based End-User Development", *CACM*, vol. 47, no. 9, pp. 43-46, Sep. 2004.

[17] T.A. Sebeok, *Signs: An Introduction to Semiotics*, Toronto: University of Toronto Press, 1994.