ComiKit – programming with comic strips

Mikael Kindborg
Programmable Toys Research
Department of Computer Science
Linköping University
SE-58183 Linköping, Sweden
Email: mikki@ida.liu.se
Web: www.ida.liu.se/~mikki

Paper presented at the European Smalltalk User Group (ESUG) Thirteenth International Conference, Brussels, Belgium, August 13 to August 20, 2005.

Summary

In this text I will present ComiKit, a programming toolkit for children that is based on the visual language of comics. I will discuss educational and object-oriented design considerations for comic strip programming tools, and elaborate on some of the problems that are not yet solved. For some parts of the text, I will assume a general knowledge of Smalltalk, and the eToys system for children in Squeak.

1. Introduction

Children have always created their own toys, often from left-over materials that happen to be available. A child-made doll or car may not be as fancy and polished as one you would buy in a store, but nonetheless, kids often find great satisfaction and joy in making their own toys. Computer games and interactive software toys are different, however. The programming required to create your own game, even a very simple one, is just too difficult for most children to learn. The consequence is that kids are limited to the role of consumers of ready-made software.

One approach for making programming easier is to use a program representation that is similar to the runtime representation. If the source code of the program is analogous to and looks similar to what is seen on the runtime display, the mental gap between the two representations could be reduced (Smith et al. 2000).

A representation that is interesting in this respect is comics. Just 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 (McCloud 1993). In comics, the story is 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 an event strip, the first panel is a precondition for the actions in the subsequent panels. For programs that consist of interactive graphical objects, comics have the potential to describe the behaviour of the objects in a way that strongly resembles the visual result of running the program (Kindborg 2003).

2. ComiKit

ComiKit is a software tool for children that uses comic strips to program the behaviour of graphical characters. The main purpose of the system is to enable children to author interactive media and to create their own interactive games and toys. The primary user group is 8 to 12 year olds.

In ComiKit, a program is created by drawing pictures of characters and making event strips for their actions. A character can have one or several pictures, and events can be used to change the picture of a character as a response to some action, for example, a character could become happy when meeting someone or something it likes. Examples of preconditions are: touching another character, having a particular picture, pressing a keyboard key. Examples of actions are: changing the picture of a character, moving, deleting a character. The tool has a play area, where programs are running, and an editor for drawing pictures and programming comic strips events. Figure 1 shows the editor for event strips. Figure 2 shows an example of the play area.

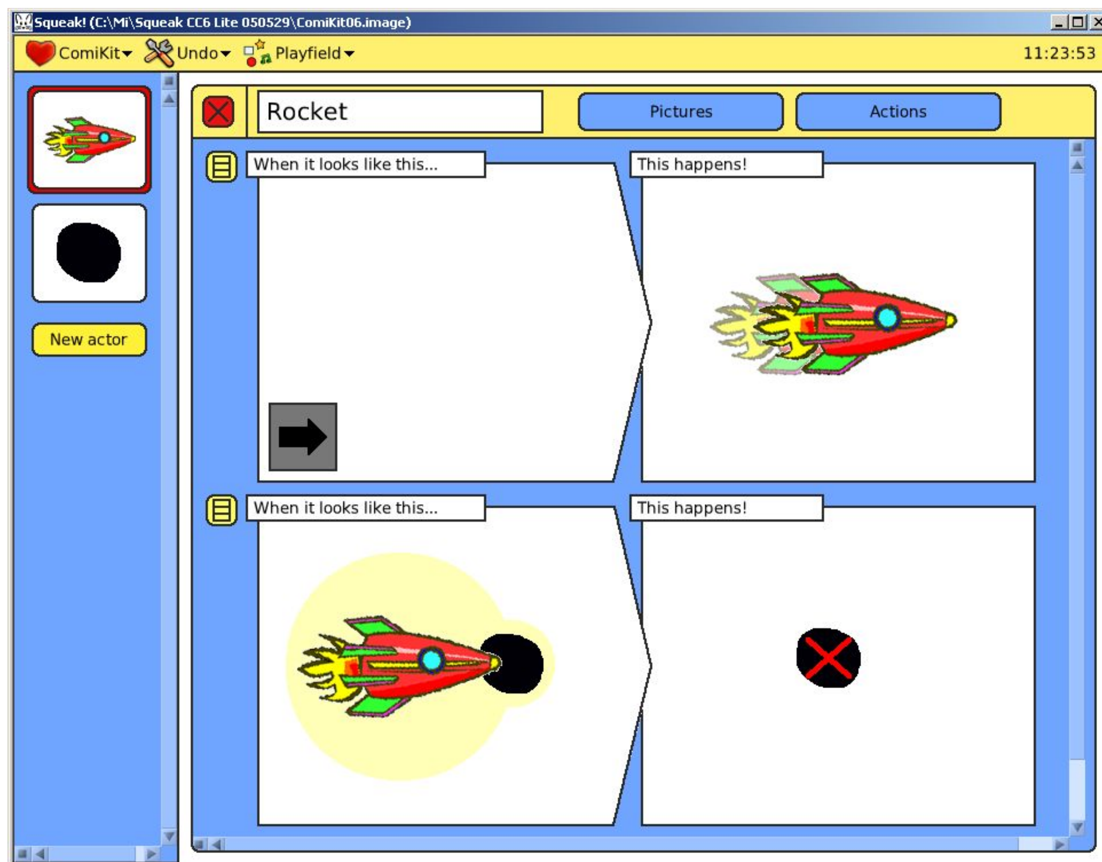

Figure 1. Comic strip events. *The first event strip has the precondition that the right keyboard arrow should be pressed, and an action that moves the rocket to the right. Note the comic-book style ghost image used to visualise the motion. (The rocket has four pictures, one for each direction right/left/up/down. In total four strips are needed to control the rocket with the arrow keys.) The second strip has the precondition that the rocket touches (collides) with an asteroid (the dark spot), and an action that deletes the asteroid. The X-cross over the asteroid means "delete". (Note that without a generalisation mechanism, four events are needed for making the rocket destroy asteroids when moving in any direction.)*

On the play area, the player can create and move characters with the mouse, causing events in the program to trigger. Events are monitored and executed by a runtime engine (interpreter). All strips that have a precondition that matches the state on the play area are executed, the order of the strips in the program is not significant.

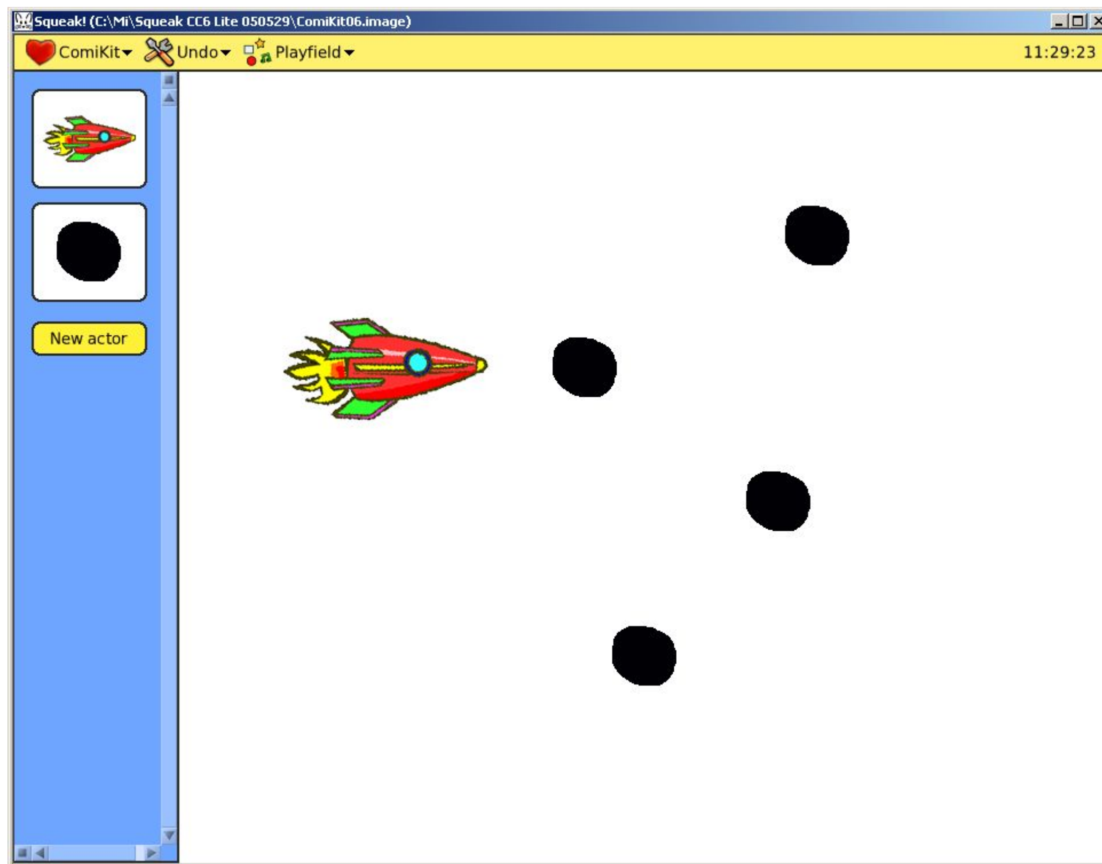

Figure 2. The play area. *The player runs a program by dragging characters from the gallery on the left into the play area. On the play area, characters can be moved with the mouse, or with keyboard keys, if that has been programmed, to cause events in the program to trigger.*

Comic strip events are similar to graphical rewrite rules, but are in many ways potentially more expressive and flexible (Kindborg & McGee 2005). Comics use a rich visual language that features pictures of domain objects (characters and various items) that are "enhanced" by contextual signs like voice bubbles and motion markers. In comics, contextual signs are used to mix iconic and symbolic signs in a way that creates a strong sense of “visual directness”. The examples shown in Figure 1 illustrate the use of contextual signs, namely, the ghost image motion marker and the X-shaped cross that signifies the delete action. Furthermore, comics use a wide variety of panel transitions that can be used by comic strip events (ibid.). In Figure 1, two different panel transitions are illustrated, from a part of the world external to the play area (the keyboard key) to a character on the screen, and from two characters meeting to one character.

The visual language of comics has a great potential for representing programs in such a way that the program looks similar to and directly maps to the action seen on the runtime display. This could help children (and adults) to create programs that feature

interactive and animated graphical objects. There are, however, several issues and design considerations that need to be addressed when designing a comic strip programming tool. In the following sections, I will discuss a number of design aspects, and at the same time provide additional details on how comic strip programming works.

3. Educational design aspects

This section discusses issues and design considerations that are related to educational aspects of comic strip programming.

How intuitive is comic strip programming?

Several prototypes of ComiKit have been tested together with 4:th and 5:th grade children in a school (Kindborg 2003). The results from these field studies showed that most children could learn how create interactive stories and simple games in a few hours. At first, I believed that familiarity with printed comics would be helpful for learning comic strip programming, but this turned out not to be the case. The children's intuitive interpretation of programs was that of a linear action sequence, which contradicted the way event strips work. The children had to learn concepts such as conditional events and non-sequential flow of control and to successfully create their own programs.

The arrow-shaped precondition panel is an attempt at giving the first panel a special status, to help learning that it is a conditional situation rather than an action. This has, however, not been tested with children yet (the prototypes used in the studies with the children used rectangular precondition panels). Regarding how to visualise the non-linear nature of event strips, this has not been experimented with yet. Possibly, some kind of representation based on cards with event strips could be used as an alternative to a sequential layout of the strips. This could for example be similar to how eToys shows scripts as “cards” in the world.

Iconic vs. symbolic representations

An iconic representation of objects in the program can help understanding what the program does, but there are many program constructs that can not be represented with iconic signs. The way comics mix iconic and symbolic signs could be used to represent program constructs such as textual/numerical object properties. Instance variables could be shown in the direct context of an iconic object, potentially making the program easier to understand.

A related aspect is the idea of learning symbolic thinking by programming. Smalltalk had the slogan “Doing with images makes symbols”, to describe how programming with graphical objects can be used to learn symbolic thinking. The eToys system in Squeak takes advantage of this idea. Comic strip programming is based on a different approach, using representations that are analogous to each other to make programming easier. The question is what the advantages and disadvantages of these two approaches are with respect to “learning how to think” and “media creation”.

One potential with the comic-book style is that mixing symbolic and iconic signs in the program representation could help bridging the understanding of iconic and symbolic representations. One example of this is given below, in Figure 3.

Turtle graphics

Figure 3 shows a mockup of how eToys-style turtle geometry could be visualised with comic-book signs. This example uses a ghost image motion marker, and illustrates how a symbolic textual representation can be shown in the context of an iconic representation of a graphical object.

A problem with representing symbolic values visually is when the value is not known. In Figure 3, numerical constants are used, and these have a direct mapping to the graphical representation. But if variables would be used for the forward motion and the rotation, there would be a problem with how to visualise this, since the values are not known. One solution could be to use example values to show how the motion could look.

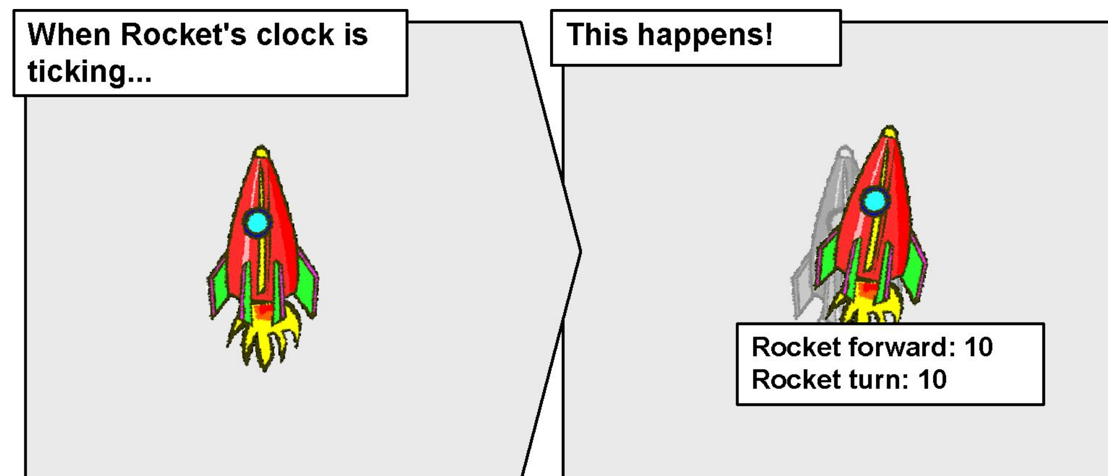

Figure 3. Turtle graphics. *The first panel contains the condition that the "clock" is ticking. The second panel gives an example of the forward motion and the rotation. Note that this motion is relative to the current heading of the rocket; when the event strip is executed the rocket will move and turn relative to its current position. The motion of the rocket could be relative to the position of the rocket in the first panel, or alternatively, the motion could be specified in the second panel by positioning the rocket and the ghost image. In that case, the rocket need not be present in the first panel; instead it could show the face of a clock ticking. The symbolic and the iconic representations of the motion could be bi-directional; changing the numerical values would change the picture, and changing the picture would change the numbers.*

Voice bubbles for textual output

The prototype systems tested in the field studies used voice bubbles for displaying text "spoken" by characters. The programs the children made focused on foreign language learning, and they created interactive stories and games in English. Typing text in bubbles worked well for these programs, and is an example of how comic book signs can be used in the runtime visualisation as well as in the program representation. Another aspect is that voice bubbles look exactly the same in an event strip as on the play area. Figure 4 gives an example of how voice bubbles were used in the programs the children created. A problem was to decide for how long a bubble should stay visible (it does not look good to leave it visible). The system computed a value based on the length of the text, but it might have been better if this could be specified by the programmer, and/or if a bubble could be closed by clicking at it.

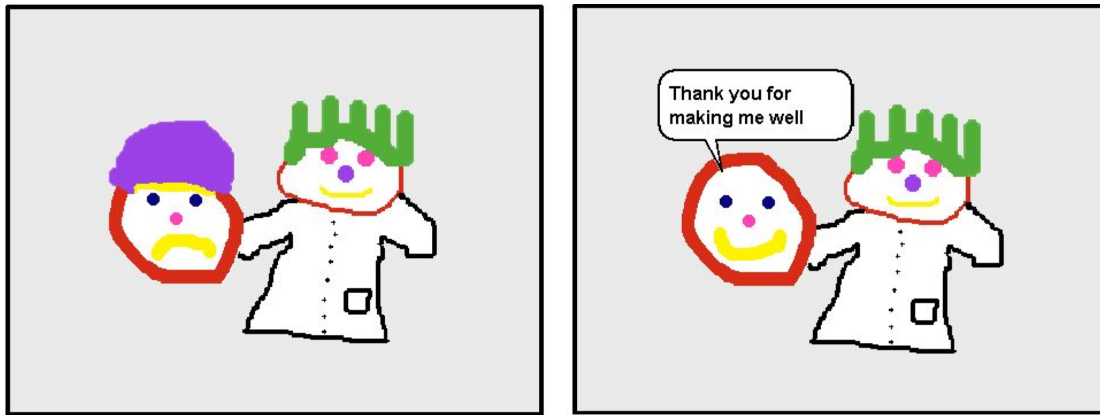

Figure 4. Voice bubbles. *This event strip was created by a girl in grade five. In the precondition panel the sad and sick-looking character meets with (touches) the doctor. In the action panel the sick character changes its picture to a happy looking face and displays the text "Thank you for making me well" in a voice bubble. The bubble will appear exactly as shown in the event when the game is played.*

One or many action panels?

The prototypes (and also the current system) had the limitation that there could only be one action panel in each event strip. This was made primarily to simplify the implementation. Most of the interactive stories the children created were designed to be played in a linear way, and when the children were asked to describe their programs, they typically told the plot of the story. Interestingly, it turned out that the stories could be played with in a much less linear way than could be anticipated at first (Kindborg 2002). The event structure with only one action panel forced the children to structure their programs into several different events. Because events can trigger in any order depending on how a game is played, it was possible for the player to experiment with the story structure and play the games in a non-linear way. Creating interactive stories and playing with them in various ways could be way for children to move from a linear narrative perspective to a non-linear rule-based perspective.

The question is what would happen if several action panels could be used? Would children create animation sequences rather than interactive events? A technical issue is how a sequence of action panels should be executed. Should there be a delay between the actions in the panels, and how should this delay be specified?

Pre-staged play area vs. empty play area

The gallery with characters is always available to the player. That means that it is the player who decides which characters to put on the play area, and how to position them. This is like having a box of toys. You can play with the toys any way you like. If the programmer could save the play area and hide the gallery, she could create games that would be more limited with regard to player freedom. Of course, there are many advantages with a predefined play area. For example, the player does not have do the work of setting up the initial game state. And it is possible to create a game that feels more like a ready-made game that you could buy in a store. The current decision for ComiKit is to be able to save the play area as part of a game, and possibly hide the gallery, but it should always be possible for the player to display the gallery, and also to view and edit characters and their events.

4. Object-oriented design aspects

This section discusses aspects of comic strip programming that are mainly related to concepts in object-oriented programming.

Classes or prototypes?

The systems tested by the children, as well as the current implementation, use a model similar to classes and instances (without inheritance). A character can be viewed as a class (a type) and many instances of that class can be created by dragging characters from the gallery onto the play area. Events belong to classes and every instance will therefore change its behaviour when the events for the class are edited. An alternative model is to use instances only, and put events directly on the instances, similar to how scripts for objects work in eToys. In eToys, an instance can also serve as a prototype, and siblings made from a prototype (using the olive handle) "inherit" the scripts from the prototype.

The question is what the benefits and drawbacks of classes and prototypes are, with respect to comic strip programs? I think that one important difference is related to the user interface, namely, having an object gallery or working directly with objects on the play area. A gallery provides a structure for objects and works as a menu for creating new instances. Working with instances/prototypes directly on the play area could feel more direct, and I have observed children who wanted to edit events for an instance. If you just want one instance with a particular behaviour, you can do that with prototypes, but when having classes, a new class needs to be created. On the other hand, when working with objects on the play area, it can be difficult to keep track of objects, and to tell the difference between prototypes and siblings, unless support for this is given in the user interface.

The self-issue

In ComiKit and in the previous prototypes, event strips belong to characters. There is always a main character that "owns" an event (this is similar to the concept of self in Smalltalk). This helps when parsing event strips and simplifies the implementation of the interpreter. At runtime, the events that belong to the class of an instance on the play area are executed. When executing an event, the interpreter first binds the instance itself to the self character in the strip. Next, the other characters in the strip are searched for on the play area and bound to characters in the event.

In the field studies, the children experienced problems with placing events on the correct character. One kind of mistake was that an event was created for the "wrong" character, for example a character that should not be part of the event. In eToys, the issue of self does not seem to be a problem, because scripts refer to named objects, and generalisation over types is not used in the same way as in comic strip events.

In some cases, it does not matter which character an event belongs to. The second event in Figure 1, for instance, could be placed either on the rocket or on the asteroid, with the same runtime result. But it would not have worked to place it on an unrelated object that is not part of the event, say a star.

The first strip in Figure 1 is an example of an event that is important to place on the right character. The self character in this strip is the rocket. Every instance of the rocket will move right when the right arrow key is pressed, because every instance

will execute this method. Placing this event on another character than the rocket would not work in the current implementation; the result would be that no rockets move, because the interpreter would fail to bind the rocket in the event to an instance on the play area.

In the prototypes used in the field studies, both panels of a newly created event strip contained the self character, and it could not be deleted. In ComiKit, a more flexible approach is taken; a new strip contains empty panels, and the programmer can add the characters she choses. Moreover, unlike previous prototypes, the self character must not be present in both panels for the event to be valid. This design change was done to make strip layout more flexible, increasing the expressiveness of comic strip events by allowing different panel transitions.

The problem with using the notion of self for event strip semantics is that this can be difficult to understand. Most children could learn this principle, but there were suggestions that it should not matter on which character events were placed, and that events should be in a central list, detached from characters. This would make strips more difficult to parse, however, and in some cases there would be several interpretations of what a strip means. Having events attached to objects does provide structure, which is an advantage with using the a self character.

One way to make the notion of self easier to understand could be to clearly designate the self character in a strip, for instance using the textual label “Me” or “I”. Part of the problem with understanding who the self character is, is lack of explicit structure, which is caused by the free-format layout of characters in a strip (the programmer can place characters in any position she wants). An clearer approach might be to use some kind of form-based layout for strip panels. A system that uses this style is Behavior Cards, an experimental visual programming tool developed by Robert Scholz at the Department of Computer Science, Linköping University. Figure 5 shows and example of an event card. Note that a structured form is used and that the self character is designated as “I”.

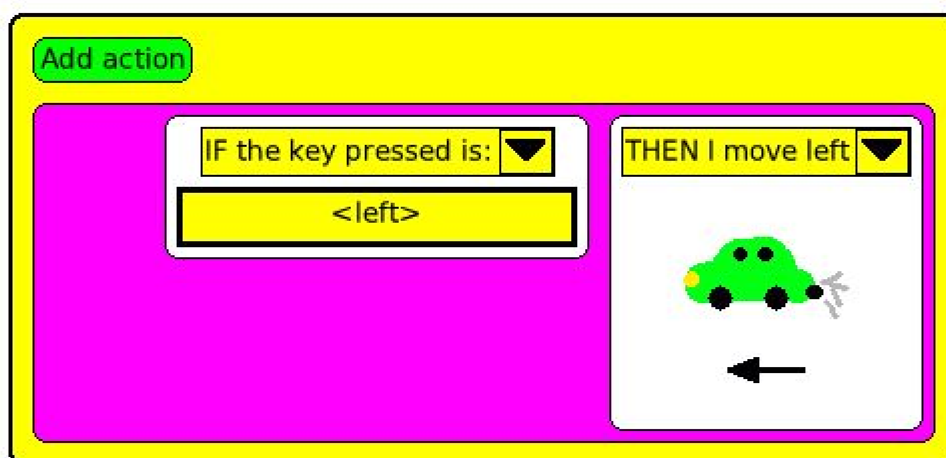

Figure 5. An event in Behavior Cards. *In this event, the car is the self character, and it moves left when pressing the left arrow key. Textual labels and a form with menu options are used to provide structure to the programmer.*

Using a structured layout for comic strip events could help solve some of the problems with ambiguity regarding how to read events and understanding the concept of a self character. However, it feels fun to place characters in panels in the way you like, and it would be sad if that feeling would be lost.

Character pictures

A less complex issue that came up in the field studies was the structure with having several pictures for a character. Characters can have one or several pictures, typically happy, sad, and so on for an animal or person, and for example flying and exploded for a space craft. A mistake made by some children was to create a picture for a new character as a picture of an existing character. For example, one child did a drawing of a telephone as one of the pictures of a doctor. In this case, the telephone had to be a separate character to make the program work.

Besides conceptual problems, a drawback with this structure is when you want to give a character a picture of another character. Some children solved this by making an event that deleted the character to be changed and created a new one at the same location. This also changed the behaviour of the character, since an instance of a another class was created. (Note the difference between changing the picture and changing the type of an object.)

An alternative to the current structure with character-centered pictures, is to have a flat structure, where characters could take on any picture the programmer wants. This is the way eToys work. While this could be easier to understand and perhaps more flexible, it does not provide a structure for organising character pictures.

Generalisation

The number of events required for a behaviour can be reduced if the look of a character can be generalised. In Figure 1, four events like the second strip are needed to program the rocket to destroy asteroids when moving in any direction (one event for each direction). This is because a picture in the precondition must match the exact same picture on the play area for the event to trigger. With picture generalisation, a character in an event would match regardless of the current picture. For the second strip in Figure 1, this means that only one event would be needed to program the asteroid destruction behaviour of the rocket. The question is how picture generalisation should be visualised? One way could be to use a grayed-out image of the first picture of a character. Another question is what should be the default? Picture matching or picture generalisation? Currently, picture matching is the default, and a menu option can be used to generalise the picture. However, it could be the reverse, generalisation being the default, with an option for choosing exact picture matching.

5. Implementation notes

ComiKit is developed in Squeak. For the prototypes used in the field studies, Java was used, but moving to a Smalltalk-based system proved to be a tremendous benefit, among other things because of the dynamic, refactoring friendly nature of Smalltalk and the expressiveness of the language. The size of the source code for the final Java version is 516KB, and the size of the current source code for the Squeak version is 274KB, with more functionality than the Java version has.

The systems parses event strips and creates an internal representation of the program code for the events. The interpreter then executes this code. An alternative would be to compile event strips to Smalltalk expressions, as is done in eToys.

A strong reason for developing in Squeak is that Smalltalk was originally designed as a tool for children, and Squeak has a lively community with people who are working with new kinds of programming tools for children.

6. Conclusion

I believe that the visual language of comics has much to offer designers of visual programming tools. A powerful presentation technique used in comics is the integration of symbolic signs into the context of iconic representations of domain objects. This technique could be used to design visual program representations that are both expressive and look similar to the runtime representation.

However, there are also many challenges and design issues that need to be addressed. Comics lack signs for expressing programming constructs like conditionals, non-linear flow of control, variables, and generalisation. Signs for such concepts need to be found or invented.

A potential future project is to integrate comic strip programming with Morphic and eToys in Squeak.

The ComiKit system can be downloaded at: http://www.comikit.com/esug/

References

Kindborg, M. (2002). Comics, programming, children, and narratives. In Proceedings of *Interaction Design and Children*, eds. Bekker, Markopoulos, Kersten-Tsikalkina, pp. 93-109. Eindhoven, The Netherlands, August 28-29.
(Available at: http://www.ida.liu.se/~mikki/comics/)

Kindborg, M. (2003). Concurrent Comics - programming of social agents by children. Linköping Studies in Science and Technology, Dissertation No. 821. Linköping University. (Available at: http://www.ida.liu.se/~mikki/comics/)

Kindborg, M., McGee, K. (2005). Comic Strip Programs: Beyond Graphical Rewrite Rules. Paper accepted for the *International Workshop on Visual Languages and Computing*, Banff, Canada, 5-7 September 2005.
(Available at: http://www.ida.liu.se/~mikki/comics/)

McCloud, S. (1993). *Understanding Comics*. New York: HarperCollins Publishers.

Smith, D. C., Cypher, A., Tesler, L. (2000). Novice Programming Comes of Age. *Communications of the ACM*, vol. 43 no. 3 (March), pp. 75-81.