Human-Computer Interaction -- INTERACT'03
M. Rauterberg et al. (Eds.)
Published by IOS Press, (c) IFIP, 2003, pp. 152-159

Programmorphosis: a Knowledge-Based Approach to End-User Programming

Andri Ioannidou

AgentSheets, Inc., 6560 Gunpark Dr., Boulder, CO 80301, USA

andri@agentsheets.com

Abstract: Whereas sophisticated computer users can exercise more control in what they are exposed to and produce computational artifacts, technologically challenged end-users assume a more passive role in the information society. In order to construct such artifacts, typically some programming knowledge is necessary. Although learning how to program is not impossible for these end-users, it is usually quite difficult and uninteresting. This paper describes Programmorphosis, a multi-layered approach to end-user programming, which, at the highest level, enables novice end-user programmers to define behaviors of interacting agents in a high-level abstract language. Behavior templates are used to structure domain concepts in declarative knowledge bases. Specifying behaviors is achieved by altering behavioral parameters in a wizard environment that subsequently generates lower-level executable code using information in procedural knowledge bases. With Programmorphosis, the programming process transforms into modification and customization of reusable templates and therefore becomes more manageable for novice end-user programmers.

Keywords: end-user programming, high-level languages, procedural knowledge bases, declarative knowledge bases, agent-based simulations, educational technology.

1 Introduction

There is a big discrepancy between computer users who can create computational artifacts and those who cannot due to a lack of programming skills. This division has rendered the majority of citizens of todayÕs information society consumers of information and computational artifacts. Whereas sophisticated computer users and developers have the ability to exercise more control in what they are exposed to, the technologically challenged end-users remain at the mercy of information producers and thus assume more passive roles. Incentives for these consumers to become producers exist: they, too, have a need to communicate ideas, realize their own designs, build their own artifacts according to their own specifications, and not just use someone elseÕs creations. Being able to control information delivery and produce computational artifacts can not only give users a higher control over what is directed to them, but also give them control over what is created and then communicated to others. Moreover, empowering end-users to create computational content themselves can be an educationally effective and rewarding activity. Learning, according to advocates of constructionism, is an active construction of knowledge as well as a social activity, as people communicate ideas and share knowledge via external artifacts (Papert & Harel, 1993).

An extensive range of computational possibilities is available, but most of them only to those who have the necessary skills to create their own artifacts. Granting such creative power to the masses requires programming. Although not completely impossible to learn, programming can be difficult for end-users (Guzdial, 1994; Pane & Myers, 1996). More importantly, however, it is an activity in which end-users do not want to engage (Mostršm, 2002). Non-programmers are not interested in programming per se, but wish to use programming as a means to a different end Ð for example to create filters for their email, to manage their finances, or to create animations or interactive simulations.

Various end-user programming paradigms (Nardi, 1993) have been introduced to ease the programming process, but despite recent developments, building computational artifacts from scratch is still difficult for most end-users. The level of current end-user programming approaches (e.g., Visual Basic, Flash, JavaScript, Natural Programming (Pane & Myers, 1996; Pane, Ratanamahatana, & Myers, 2001), Visual AgenTalk (Repenning & Ambach, 1996)) is too low for most end-users to benefit from creating applications or computational artifacts. Learning to program a computer in such a way that the skill can make a meaningful contribution to oneÕswork is not an easy task (Lewis & Olson, 1987). Pragmatically speaking, end-users do not have the time to devote to get to that level of proficiency in programming. They generally require approaches with more immediate results. The high cost/benefit ratio (Rader, Cherry, Brand, Repenning, & Lewis, 1998) of programming stemming from such pragmatic concerns as the time required to master a programming language keep end-users away from programming – and therefore from the benefits it can afford.

Visual languages constitute one approach to make programming easier and more accessible to end-users. However, not all problems and constructs lend themselves to visual representations. Moreover, general-purpose visual languages suffer from the same drawbacks as traditional general-purpose programming languages: they include low-level programming primitives (even if they are visually represented) and they lack domain orientation. Domain-oriented programming paradigms help significantly because they bring the programming and domain worlds closer together. But even if the domain-oriented constructs of such languages help, they are too narrow, because they focus on one specific domain and do not easily lend themselves to reuse or adaptation to other domains.

Reusing existing resources and combining them in new artifacts is yet another method for scaffolding the process of programming. Reuse traditionally happens at the level of source code, which is very useful, but reuse at a higher level – at the conceptual level – can also be beneficial. Conceptual reuse is concerned with more abstract, higher-level representations, such as genres or patterns – generalized behaviors that are observed repeatedly in multiple applications. Moreover, reuse of an existing resource "as–is" is typically not sufficient; instead, customizations for individual needs are necessary. Adaptation is a difficult phase in the reuse process because it is the least supported one: reusers must not only understand the reusable code or resource but also figure out what they need to change.

The end-user programming paradigms and reuse approaches mentioned above have made significant progress toward making programming available to end-users. However, they still do not provide end-users with the tools necessary for programmability – the ability to create programs *easily*. Programming – even end-user programming – is a process of synthesis (Lewis & Olson, 1987), and as such it is difficult (Pane & Myers, 1996). For end-users, however, programming should be transformed from a process of synthesis into a process of customization and modification.

2 Programmorphosis

This paper presents *Programmorphosis* (Ioannidou, 2002), a new end-user programming approach that replaces synthesis (Lewis & Olson, 1987) with modification and customization in the process of programming, and replaces high-level specifications with low-level representations in the program itself. By definition, transformation in Programmorphosis, is twofold:

- ***Metamorphosis of the programming process***: The programming process is transformed from a synthesis task to a customization and modification task.
- ***Metamorphosis of the program***: The program itself is transformed from the high-level, template-based, parameter-based specification to a low-level language primitive specification through program generation.

Figure 1 explains how Programmorphosis is achieved:

1) The template designer creates the *declarative knowledge base,* which describes *what* the particular behavior template is, using a meta-level representation language. The declarative knowledge base contains specifications of the templates that capture semantic information about the template, descriptions of the functionality of the template, what other templates it is referencing or needs to work with, documentation and examples of usage. It also contains semantic information that can be used to generate the high-level user interface.

2) The template designer creates the *procedural knowledge base*, which contains information on *how* the transformation from the high- to the low level behavior is done for the particular template. Essentially, it provides the mapping from the high-level representation of the behavior to the lower-level executable code and can therefore be thought of providing the compilation mechanism for the high-level language used in the template.

3) The template designer links the declarative and procedural knowledge bases by including pointers from the declarative to the procedural knowledge base to enable the system to use the appropriate procedural knowledge for a given template.

4) The semantic information included in the declarative knowledge base is used by the *wizard generation* system to automatically generate the high-level user interface. The wizard high-level interface is not hard-coded, but is completely defined at a high, semantic level in the declarative knowledge base and therefore its generation can be automated, using mechanisms built-in the wizard generation system that allow the rendering of interface components from semantic specifications.

5) When the end-user interacts with the high-level wizard interface and defines behaviors by specifying parameters in the template, the program generation system translates these high-level specifications to the low-level executable behavior,

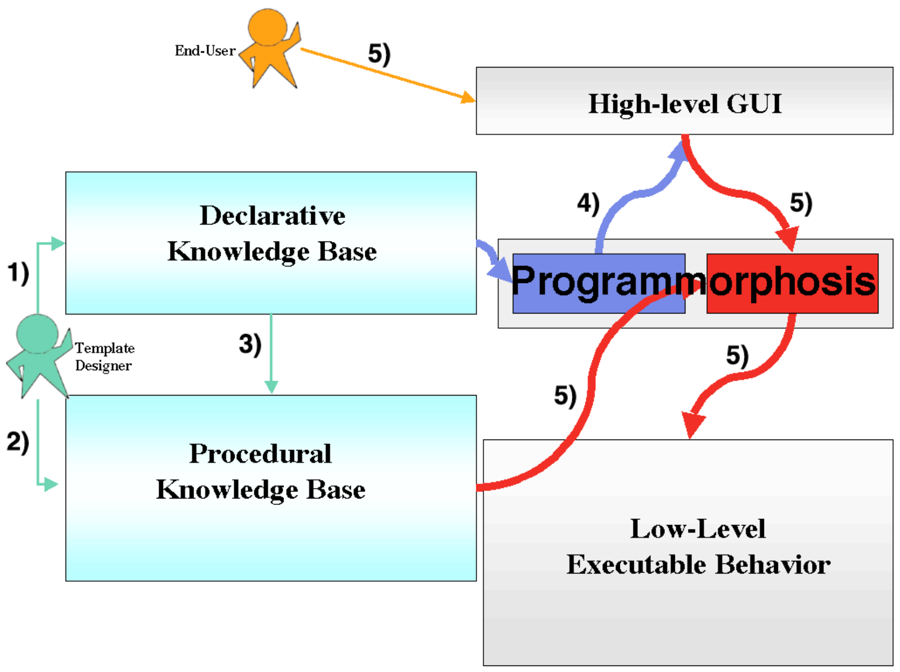

End-User
5)
High-level GUI
Declarative Knowledge Base
4)
5)
1)
Programmorphosis
Template Designer
3)
5)
5)
2)
Procedural Knowledge Base
Low-Level Executable Behavior

Figure 1: High-level overview of the Programmorphosis approach: Template designers create declarative and procedural knowledge bases that are used to transform the programming process to a modification of templates. End-users interact with the high-level interface to specify behaviors and the system generates the executable behaviors.

using information from the appropriate procedural knowledge base for the given template.

What the template designer does, namely, specifying the declarative and procedural knowledge bases, constitutes the *programming transformation* part of Programmorphosis (1-4 in Figure 1) by providing the essential components needed to transform the process of programming from a process of synthesizing low-level language primitives to a process of modifying and customizing existing high-level parameter-based templates.

The part of the system that the end-user interacts with, including the high-level user interface and the program generation functionality that uses the procedural knowledge base, constitutes the *program transformation* part of Programmorphosis (5 in Figure 1), which transforms the high-level parameter-based specifications of behavior in the domain-oriented templates to the low-level executable code.

Programmorphosis is an end-user programming approach by which users use a collection of *reusable behavior templates* and customize them through *behavioral parameters*. They do this with the help of a *wizard* that provides *sustained support* of the customization and modification process and also *generates executable program* by transforming this high-level specification to a low-level code. The wizard adds yet another programming level that is more accessible to novice end-user programmers, rendering a multi-layered programming architecture. These aspects of Programmorphosis are explained in the following sections.

2 .B1ehavioral Templates and Parameters

Templates used in Programmorphosis afford behavior genre reuse, which provides cognitive and technical access to behavior building blocks at different granularities such as entire simulations and simulation components (agents) and reuse mechanisms to utilize them in constructing new computational artifacts. In essence, reuse is occurring through the identification, instantiation, and customization of behavior genres templates.

The programming transformation process in Programmorphosis entails abstracting reusable aspects of behaviors to render them modifiable at a high level via parameterized behavior authoring. A higher level of authoring is achieved by designing behaviors with several behavior-altering parameters that can, among other things, control numerical input to the entity that can alter its behavior or even control behaviors that specify interactions with other entities, namely agents.

Depending on the degree of structure the template author wants to impose, behaviors can have a variable number of parameters. Providing more structure may mean that most of the underlying behaviors are set and a small fraction of the behavior is controllable via parameters. At the other extreme, complete flexibility can be given to the user by providing as many parameters as possible for complete control of the behavior.

2.2 Knowledge Bases

To achieve both the program and programming transformation aspects of Programmorphosis, information needs to exist to specify *what* the program at the high-level is (i.e. what the template is about) and *how* the transformation is performed from the high- to the low-level behavior representations. This information resides in two distinct knowledge bases.

- The *declarative knowledge base* contains information on what the template is (i.e. description in domain terminology, contents) and what its interface looks like (i.e. semantic information that can be used to render the high-level wizard interface).
- The *procedural knowledge base* contains information on how to transform the behavioral parameters in the high-level template interface into executable low-level code. Therefore, the procedural knowledge base assists program generation.

2.3 Sustained Wizard Support

Tools that support Programmorphosis can be software wizards assisting end-users to retrieve, evaluate, and most importantly customize a reusable behavior template. The wizard should provide a step-by-step walkthrough of the complete specification of the interacting entities, to alleviate some of the issues involved in the problem-solving phase of the programming task, such as the division of the overall task into manageable sub-tasks. But the support should not stop there. The template-based wizards for defining behaviors need to go beyond the traditional generate-only wizard. Behavior needs to be both *generated* and *edited* using the high-level wizard interface.

Providing a generate-only wizard model is inadequate for novice end-user programmers because it provides support only for the initial phases of the programming task. This would mean that any required editing of the generated behavior would have to be done at the lower level, which presents the same issues as synthesizing code from scratch. It is easier to read and understand existing code and perform minor modifications to it than to synthesize code from scratch, but even if that code is at a low level, novice end-users will still be somewhat perplexed in terms of what needs to be changed, as well as where and why.

2.4 Program Generation

With Programmorphosis, the end-user programmer supplies the program in a high-level specification language, namely the parameterized templates. This happens at an abstract level similar to the specification level in traditional programming languages. In object-oriented design and programming, professional programmers use specification methods such as UML (Object Management Group, 2002) to capture abstract, high-level design of their systems. Some specification tools then offer code generation in the form of skeleton programs that capture the basic structure of the high-level objects. However, in the case of novice end-user programmers, generating non-functioning code would not be sufficient. Instead, the wizard translates the high-level specification into complete, executable low-level code. It is extremely important that this automatically generated low-level code be immediately executable. The immediate feedback given to the user by executing the generated code affords a shorter turnaround in the modify-generate-and-test cycle.

Low-level representations generated from high-level specifications are not sufficient, should the system allow its users to return to the high-level representation at a later stage. Meta-information generated by the wizard is essential to provide the means for end-users to revisit the high-level representation for behavior inspection and editing purposes.

2.5 Multi-layered programming

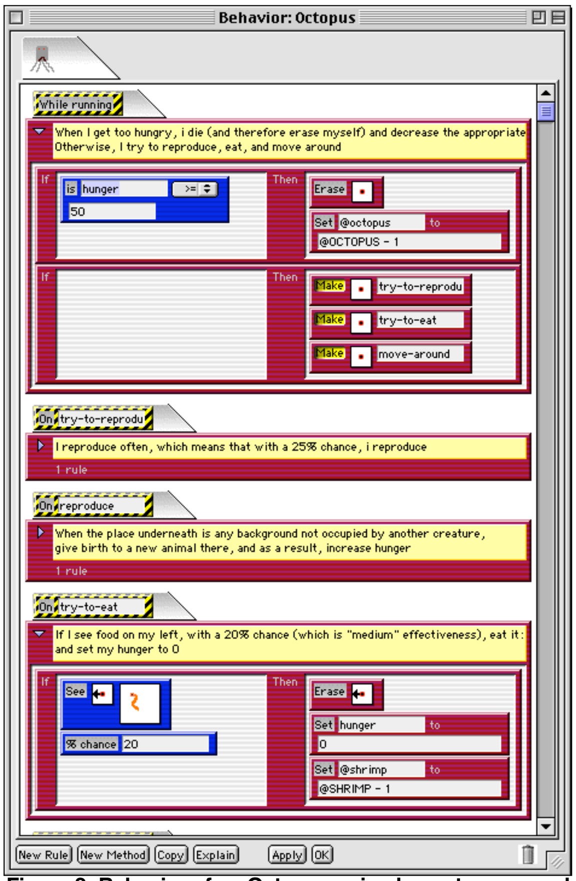
The high-level behavior specification language in Programmorphosis essentially adds a layer in the programming process. The combination of high-level wizards and program generation allows for a multi-layered programming approach that enables a wide range of end-users to do the programming because multiple levels of abstraction address the different needs of the various levels of ability of end-user programmers.

Although the Wizard level does not provide much expressiveness, multi-layer programming allows the user to move to the lower level and therefore gain the expressiveness back, if and when this is necessary. Within the templates of the Behavior Wizard, not many changes other than parameter changes are allowed. Therefore, changes not foreseen by the template designer are difficult to make at that level. The lower level, however, allows for any modification of the generated behavior.

3 Programmorphosis in a Simulation-Authoring Tool

This section provides an example of the instantiation of the Programmorphosis approach in the AgentSheets simulation-authoring tool. AgentSheets (Ioannidou & Repenning, 1999) is an agent-based simulation-authoring tool that enables end-users to build their own interactive simulations. It features an end-user programming language. Visual AgenTalk, which allows the composition of agent behaviors with if-then rules (Figure 2).

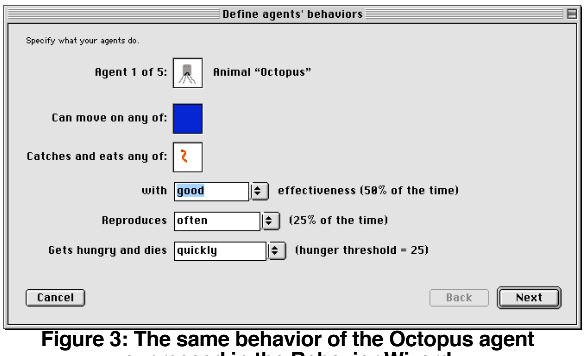
Research conducted in classrooms with the AgentSheets simulation-authoring tool (Cherry, Ioannidou, Rader, Brand, & Repenning, 1999; Rader et al., 1998; Zola & Ioannidou, 2000) has produced exceptional results along different dimensions (including content learning, social dynamics, and computer literacy), but at the same time pointed out the difficulty of programming for end-users in those contexts. This research has found that although children can describe in a general way how objects should behave, they often face major challenges when trying to translate their ideas into workable computer programs. Program composition – the ability to assemble the language primitives to achieve the desired effects – is challenging for users because they have a difficult time mapping their own ideas into rules (Rader et al., 1998). Moreover, programming concepts understood by professional programmers, such as variables, procedural abstractions, procedure calling/message passing, rule priority, and program control flow constructs are not generally familiar to students and end-users.



```
Behavior: Octopus

While running
When I get too hungry, i die (and therefore erase myself) and decrease the appropriate
Otherwise, I try to reproduce, eat, and move around
If                              Then
  is hunger >= 50                 Erase
                                  Set @octopus to @OCTOPUS - 1
If                              Then
                                  Make try-to-reprodu
                                  Make try-to-eat
                                  Make move-around

On try-to-reprodu
I reproduce often, which means that with a 25% chance, i reproduce
1 rule

On reproduce
When the place underneath is any background not occupied by another creature,
give birth to a new animal there, and as a result, increase hunger
1 rule

On try-to-eat
If I see food on my left, with a 20% chance (which is "medium" effectiveness), eat it:
and set my hunger to 0
If                              Then
  See                             Erase
  % chance 20                     Set hunger to 0
                                  Set @shrimp to @SHRIMP - 1
```

Figure 2: Behavior of an Octopus animal agent expressed in AgentSheets Visual AgenTalk

The Programmorphosis approach attempts to elevate the task of programming interacting agent behaviors, in the case of AgentSheets, from a task of synthesis to one of modification and customization. The Behavior Wizard (Ioannidou, 2002) was added to AgentSheets to accommodate this. For example, the behavior of an Octopus animal agent for an ecosystem simulation would be represented in Visual AgenTalk as shown in Figure 2, with if-then rules defining eating, mating, and moving behaviors. In the Behavior Wizard, the user would specify the same behavior by manipulating parameters such as prey, hunting effectiveness, reproduction rate, as shown in Figure 3.



Define agents' behaviors

Specify what your agents do.

Agent 1 of 5: Animal "Octopus"

Can move on any of:

Catches and eats any of:

with good effectiveness (50% of the time)

Reproduces often (25% of the time)

Gets hungry and dies quickly (hunger threshold = 25)

Figure 3: The same behavior of the Octopus agent expressed in the Behavior Wizard

Figure 4 explains in more detail the instantiation of Programmorphosis in AgentSheets.

1) The declarative knowledge base is represented and stored as *XML*. The declarative knowledge base for each template includes meta-information such as a description of the template behavior, agent types included in the template, parameter types and descriptions for each agent behavior.

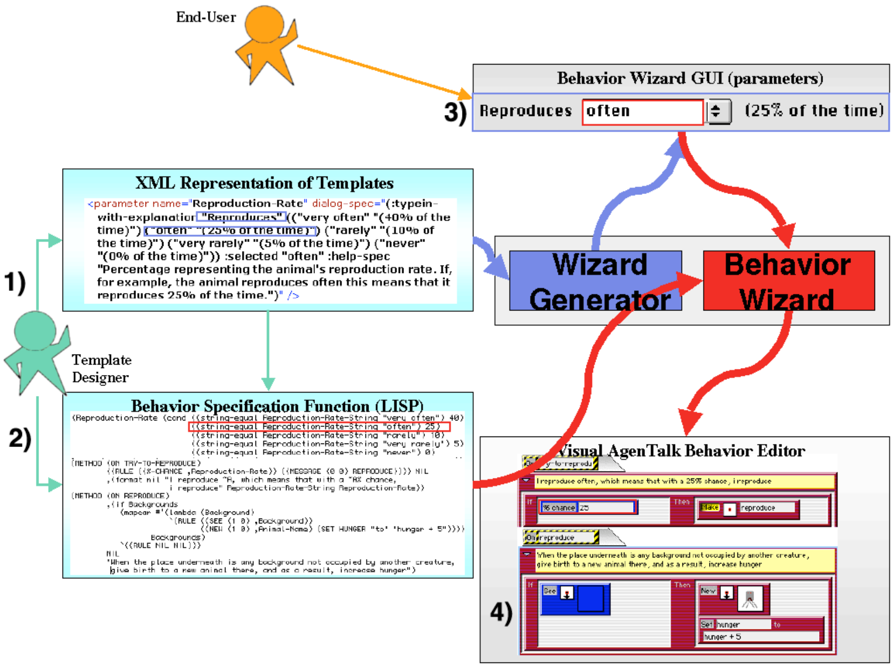
2) The procedural knowledge base consists of the *behavior specification functions* that are specified in a traditional programming language (such as Lisp) and are used to generate the low-level code from the high-level representations of the behavior in the Behavior Wizard GUI.

3) The *Behavior Wizard GUI* consists of the user interface of the templates with the user-defined parameters. These parameters use domain-oriented language and terminology and are directly manipulated to specify behaviors for each of the agents in a simulation.

4) The low-level executable behavior is in the case of AgentSheets the Visual AgenTalk behavior command form that is used both to execute an agent's behavior and represent its behavior in the lower-level level of Visual AgenTalk behavior editors.

The Wizard Generator is the functionality that generates the high-level interface using the information specified in the declarative knowledge base. Along with the mechanisms for the specification of the procedural and declarative knowledge bases it constitutes the *instantiation of the programming transformation* aspect of the Programmorphosis as was applied to AgentSheets.

The Behavior Wizard, the *instantiation of the program transformation* aspect of Programmorphosis, is the combination of the high-level GUI and functionality that uses the procedural


End-User
Behavior Wizard GUI (parameters)
3)
Reproduces often (25% of the time)
XML Representation of Templates
1)
Template Designer
2)
Behavior Specification Function (LISP)
Wizard Generator
Behavior Wizard
Visual AgenTalk Behavior Editor
4)

```
<parameter name="Reproduction-Rate" dialog-spec="(:typein-
  with-explanation "Reproduces" (("very often" "(40% of the
  time)") ("often" "(25% of the time)") ("rarely" "(10% of
  the time)") ("very rarely" "(5% of the time)") ("never"
  "(0% of the time)")) :selected "often" :help-spec
  "Percentage representing the animal's reproduction rate. If,
  for example, the animal reproduces often this means that it
  reproduces 25% of the time.")" />
```

```
(Reproduction-Rate (cond ((string-equal Reproduction-Rate-String "very often") 40)
                         ((string-equal Reproduction-Rate-String "often") 25)
                         ((string-equal Reproduction-Rate-String "rarely") 10)
                         ((string-equal Reproduction-Rate-String "very rarely") 5)
                         ((string-equal Reproduction-Rate-String "never") 0)
(METHOD (ON TRY-TO-REPRODUCE)
        ((RULE ((%-CHANCE ,Reproduction-Rate)) ((MESSAGE (0 0) REPRODUCE)))) NIL
        ,(format nil "I reproduce ~A, which means that with a ~A% chance,
                      i reproduce" Reproduction-Rate-String Reproduction-Rate))
(METHOD (ON REPRODUCE)
        ,(if Backgrounds
           (mapcar #'(lambda (Background)
                      `(RULE ((SEE (1 0) ,Background))
                             ((NEW (1 0) ,Animal-Name) (SET HUNGER "to" 'hunger + 5"))))
                 Backgrounds)
           `((RULE NIL NIL)))
        NIL
        "When the place underneath is any background not occupied by another creature,
         give birth to a new animal there, and as a result, increase hunger")
```

On try-to-reproduce
I reproduce often, which means that with a 25% chance, i reproduce
If % chance 25 Then Make reproduce

On reproduce
When the place underneath is any background not occupied by another creature, give birth to a new animal there, and as a result, increase hunger
If See Then New
Set hunger to hunger + 5

Figure 4: Overview of the instantiation of Programmorphosis in AgentSheets.

knowledge base to transform the high-level specification of behavior to the low-level implementation of the program.

4 End-User Experiences

A user experiment to test Programmorphosis took place in a Biology class at New Vista, a local high school that is fairly receptive to innovative educational technology research ideas.

In collaboration with the teacher, we chose the task for the high school students to create a simulation of a balanced ecosystem. The ecosystems domain is a fairly difficult domain to explore and creating a sustainable ecosystem is an intrinsically challenging task. Creating dynamic and interactive ecosystems is an instance of appropriate usage of computers for learning in educational settings. Just reading about ecosystems in a book does not give students the opportunity to experience the degree of interconnectedness of animals and food webs in a dynamic system. Whereas it is difficult to picture all the interactions and the effects of species populations on each other as a thought experiment, computer-based interactive simulations can achieve that very easily, providing a way for students to think about ecosystems in terms of system dynamics.

In this experiment, which was conducted in five two-hour sessions over the course of two weeks, 5 groups of students (2 students per group) used AgentSheets enhanced with the Behavior Wizard to complete the task.

Results from the experiment illustrated that the Programmorphosis approach enables novice end-user programmers to create a complete simulation in a timely manner appropriate for use in classrooms. Using the behavior Wizard, all students had created a complete, executable simulation in less than an hour. Their simulations varied in their ecosystem theme, number of species included, and sustainability. One example is the simulation of an ocean ecosystem shown in Figure 5.

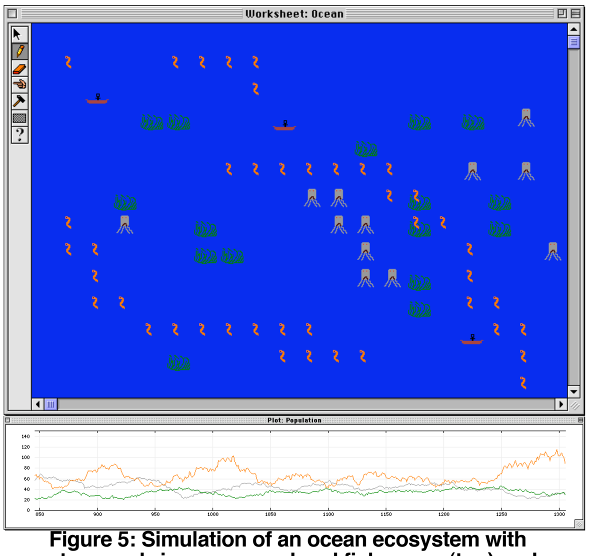


Figure 5: Simulation of an ocean ecosystem with octopus, shrimp, seaweed and fishermen (top) and simulation plots (bottom).

Moreover, the high-level programming enabled by the Programmorphosis approach affords focus on the domain instead of programming issues. After creating their simulations, the students spent the rest of the time dedicated to this experiment balancing their animals' characteristics to create a sustainable ecosystem, which is exactly what they should be spending their time on – not in struggling with programming. The time spent on running a simulation and discussing, hypothesizing, and changing parameters according to hypotheses can be considered time spent on task. Data analysis of computer logs kept by the software and discourse analysis performed on student conversations video- and audio- taped indicated that the biggest percentage of the time was spent on task and not on programming.

Finally, students using the Behavior Wizard-enhanced AgentSheets to create and balance their ecosystems learned about the domain as summarized in Table 1.

5 Conclusions

Related work on educational systems and particularly simulation systems include StartLogo (Resnick, 1994, 1996) and StarLogo^T (Wilensky, 2000) which provide parameterized simulations (microworlds) and high-level interfaces with sliders, buttons and number entries for controlling those parameters. Unlike the Wizard in Programmorphosis, these high-level interfaces are only used to control, not build a simulation from scratch. Whereas authoring is available, it is offered in the form of a textual programming language (Logo) which is not entirely intuitive for novice end-users (Cohen, 1990).

LiveWorlds (Travis, 1994) is a graphical environment designed to support programming with active objects. It offers novice users a world of manipulable objects, with graphical objects and elements of the programs that make them move integrated into a single framework. The programming philosophy rests on the premise that behaviors need to be as accessible as the objects themselves. The system thus makes behavior parameters easily modifiable. Whereas this approach makes programming more accessible to novice end-users, the level of the programming language is often not appropriate for those users. The multi-layered programming approach in Programmorphosis decouples lower-level programming from such activities as modeling, which is an important computational literacy (diSessa, 2000; President's Committee of Advisors on Science and Technology, 1997). To make modeling accessible to end-users, it should not be equated to programming or be dependent on programming knowledge.

Even though programming can be a rewarding activity with a potentially big value for learning skills such as problem solving, end-users do not care much for programming per se. They are typically interested in programming as a vehicle to achieve goals such as learning domain concepts in a particular content area. It is therefore important, especially for educational technology, to provide programming interfaces that are intuitive and easy to use, and, most importantly, that keep end-user programmers focused on the domain task they are trying to solve and not get them distracted by programming details.

Programmorphosis achieves that goal by providing a high-level programming paradigm of modification and customization of behavioral templates by altering behavior-controlling parameters and low-level executable code generation. With programming details hidden, the accidental complexity of programming activities such as ecosystem modeling is eliminated, and users can focus on the domain and deal with intrinsic complexities of the task at hand.

However, the high-level end-user programming approach featured in Programmorphosis has its trade-offs. The high-level wizard interface provides scaffolding while sacrificing some expressiveness. For novice end-user programmers, it may be beneficial to sacrifice some expressiveness and control to gain structure and support the programming activity. In time, the system should provide self-disclosure mechanisms (DiGiano & Eisenberg, 1995) that would enable users to make the mapping from high- to low-level representations. Revealing the lower-level programming could potentially enable users to learn some programming concepts and grant them more control and the ability to express more.

In educational settings, the Programmorphosis approach and the Behavior Wizard system enable end-users to explore domain concepts at the tool level. However, learning cannot be expected to happen exclusively via interactions with the computational tool. Curricular support in the form of surrounding material and activities as well as timely and guided instruction are necessary to successfully integrate constructionist media in educational settings. With constructivist educational systems "emphasis [should

What students claimed they knew before	**What students claimed they learned**	**What researchers think students learned**
Existence of predators and prey Trophic pyramid Primary Producers Primary Consumers Secondary Consumers There should be a balance There could be imbalance	Small changes in characteristics can have a big effect on the species itself or more than one species Have more of a grasp of population interactions Food chains/webs One species can wipe out another population Surprised by how fast something can reproduce and how fast it can die out It's really hard to get something balanced	Applied and tested existing content knowledge Interconnectedness and interdependency of species Complexity of ecosystems Fragility of ecosystems Modeling skills Experimentation skills

Table 1: Students learning with the Behavior Wizard.

be] on learning rather than teaching, and on facilitative environments rather than instructional goals" (Collins, Neville, & Bielaczyc, 1999). Although the emphasis is more on learning than on teaching, this does not mean that the role of the teachers has somehow been diminished. On the contrary, the role of teachers is now much more important. Teachers have to create the appropriate context and structure in which the technology is going to be used, be familiar with ways of introducing technology to the class, and be willing to learn and explore the technology along with their students. Therefore, designers of educational technology need to support teachers in their attempt to apply their technologies in classrooms and work with them to provide environments that facilitate learning. For practices to improve, we need to address pragmatic issues, including curriculum constraints, deficiencies of up-to-date equipment, and limited resources to support teachers using our tools.

References

Cherry, G., Ioannidou, A., Rader, C., Brand, C., & Repenning, A. (1999). *Simulations for Lifelong Learning*. Paper presented at the Proceedings of National Educational Computing Conference (NECC), Atlantic City, NJ.

Cohen, R. S. (1990). Logo in the Primary Classroom: Should Simplified Versions Be Used? *The Computer Teacher*(April), 41-43.

Collins, A., Neville, P., & Bielaczyc, K. (1999). *Toward a Design Theory of Media for Education*, http://www.apc.src.ncu.edu.tw/apc/allanmedia.htm

DiGiano, C., & Eisenberg, M. (1995). *Self-disclosing Design Tools: A Gentle Introduction to End-User Programming*. Paper presented at the Designing Interactive Systems, Ann Arbor, MI.

diSessa, A. (2000). *Changing Minds: Computers, Learning, and Literacy*. Cambridge, MA: MIT Press.

Guzdial, M. (1994). Software-Realized Scaffolding to Facilitate Programming for Science Learning. *Interactive Learning Environments, 4*(1), 1-44.

Ioannidou, A. (2002). *Programmorphosis: Sustained Wizard Support for End-User Programming*. Ph.D. Thesis, Department of Computer Science, University of Colorado, Boulder.

Ioannidou, A., & Repenning, A. (1999). End-User Programmable Simulations. *Dr. Dobb's* (302 August), 40-48.

Lewis, C., & Olson, G. (1987). Can Principles of Cognition Lower the Barriers of Programming? In G. M. Olson, S. Sheppard & E. Soloway (Eds.), *Empirical Studies of Programmers: Second Workshop* (pp. 248-263). Norwood, New Jersey: Ablex Publishing Corporation.

Moström, J. E. (2002). *Using Concurrent Constructs in an Authoring Environment for Teaching*. Umeå Universitet, Umeå.

Nardi, B. (1993). *A Small Matter of Programming*. Cambridge, MA: The MIT Press.

Object Management Group. (2002). *Unified Modeling Language (UML), version 1.4*, http://www.omg.org/technology/documents/formal/uml.htm

Pane, J. F., & Myers, B. A. (1996). *Usability Issues in the Design of Novice Programming Systems* (Technical Report No. CMU-CS-96-132). Pittsburg, Pennsylvania: School of Computer Science, Carnegie Mellon University.

Pane, J. F., Ratanamahatana, C. A., & Myers, B. A. (2001). Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies, 54*, 237-264.

Papert, S., & Harel, I. (Eds.). (1993). *Constructionism*. Norwood, NJ: Ablex Publishing Corporation.

President's Committee of Advisors on Science and Technology. (1997). *Report to the President on the Use of Technology to Strengthen K-12 Education in the United States*

Rader, C., Cherry, G., Brand, C., Repenning, A., & Lewis, C. (1998). *Principles to Scaffold Mixed Textual and Iconic End-User Programming Languages*. Paper presented at the Proceedings of the 1998 IEEE Symposium of Visual Languages, Nova Scotia, Canada.

Repenning, A., & Ambach, J. (1996). *Tactile Programming: A Unified Manipulation Paradigm Supporting Program Comprehension, Composition and Sharing*. Paper presented at the Proceedings of the 1996 IEEE Symposium of Visual Languages, Boulder, CO.

Resnick, M. (1994). *Turtles, Termites and Traffic Jams: Explorations in Massively Parallel Microworlds*. Cambridge, MA: The MIT Press.

Resnick, M. (1996). Beyond the Centralized Mindset. *Journal of the Learning Sciences, 5*(1), 1-22.

Travis, M. (1994). *Recursive Interfaces for Reactive Objects*. Paper presented at the Proceedings of CHI'94, Boston, MA, April 24-28.

Wilensky, U. (2000). Modeling Emergent Phenomena with StarLogoT. *@CONCORD.org, Winter 2000.*

Zola, J., & Ioannidou, A. (2000). Learning and Teaching with Interactive Simulations. *Social Education: the Official Journal of National Council for the Social Studies, 64*(3), 142-145.