Making Programming more Conversational

Alexander Repenning
AgentSheets Inc.
Boulder 80301, Colorado, USA
alexander@agentsheets.com

***Abstract* – Accelerated by the Do-It-Yourself mindset of the Web 2.0 culture, end-user programming—programming by end users with limited or even no formal programming background—is growing rapidly. Especially in educational settings, children are exposed to computational thinking by making games, building scientific simulations and creating stories. Early educational programming languages such as Logo have made programming substantially more accessible to end-users. More recent approaches include visual programming with a drag-and-drop style of programming that makes it nearly impossible to compose syntactically incorrect programs. However, as the syntactic challenges of end-user programming are gradually fading into the past, the new frontier of semantic programming support emerges. This demonstration introduces Conversational Programming, a system to make programming more conversational. A conversational programming agent runs programs one step into the future in order to help end-users visualize discrepancies between the programs they intended to write and their actual programming results.**

Keywords – Game design; computational thinking; debugging; end-user programming; visual programming.

I. TOWARDS CONVERSATIONAL PROGRAMMING

In programming, the interaction between the programmer and the programming environment is typically asymmetrical and often limited to syntactic feedback regarding programs that are malformed. Miss one semicolon in a C program and the program may no longer work at all. A programmer may spend a considerable amount of effort on writing a program before the programming environment provides meaningful feedback.

One way to simplify programming would be to make the communication process between the programmer and the programming environment more symmetrical with the goal of aiding debugging. But just how can one conceptualize debugging? Pea [1] describes the process of debugging as:

> *systematic efforts to eliminate discrepancies between the intended outcomes of a program and those brought through the current version of the program.*

A number of programming approaches, including programming by example [2, 3] and natural programming [4], try to systematically reduce these discrepancies by having programmers demonstrate actions on concrete examples or by providing programming languages that more closely resemble the way users with no programming background tend to think about certain problems. The notion of *conversational programming*, introduced in this paper, provides a different approach that employs computational agents to provide real-time semantic feedback to a programmer so that the programmer can identify discrepancies between the *intended program* and the *actual program*. The only way to provide this kind of semantic feedback is for the computer to actually execute parts of the program as written by the user. While this translates into additional computational needs for programming environments, we find that modern multi core computers have no problem handling this extra effort. More importantly, computational cycles tend to be cheaper than cognitive ones.

Conversational Programming can be conceptualized as a simple form of pair programming [5] that replaces one of the human partners with a computational agent called the *Conversational Programming Agent* (CPA). Figure 1 describes a Conversational Programming architecture.

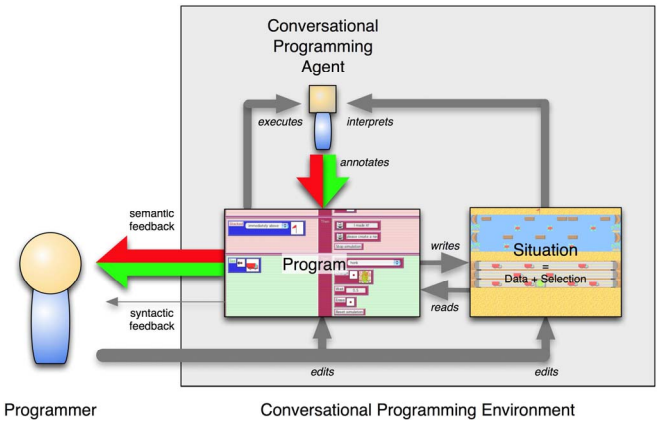


Figure 1. Conversational Programming: The programmer edits the program and edits the situation (game or simulation). A conversational programming agent executes the program, interprets the situation and annotates the program semantically.

The notion of a conversation suggest the need for a:

- ***programming partner/agent*** able to serve as another pair of eyes. Just like a partner in pair programming, this partner participates in different kinds of conversations. It can be somewhat reactive, and can simply wait for the programmer to edit the program or the situation. However, it can also be proactive and might suggest information relevant to programming tasks ahead.

- ***more symmetrical and semantic interaction*** between the programmer and the programming environment. As the programmer is editing the program, the CPA needs to be able to provide timely feedback on program semantics to the programmer in order to reveal emerging semantic discrepancies between the indented program and the actual program.
- ***shared context with a defined focus*** corresponding to a conversation topic. For instance, a programmer should be able to select an object in a game or simulation world in order to make the conversation relevant to this object and its state. This focus helps to make conversations more relevant to the programmer and also reduces computational overhead by restricting the CPA to program fragments pertinent to the conversation.

The goal of Conversational Programming is to reduce semantic discrepancies between an *intended program* and an *actual program* by using notions of conversations to make the interaction between the programmer and the programming environment more symmetrical, more timely, and more meaningful.

II. Prebugging: Proactive Debugging

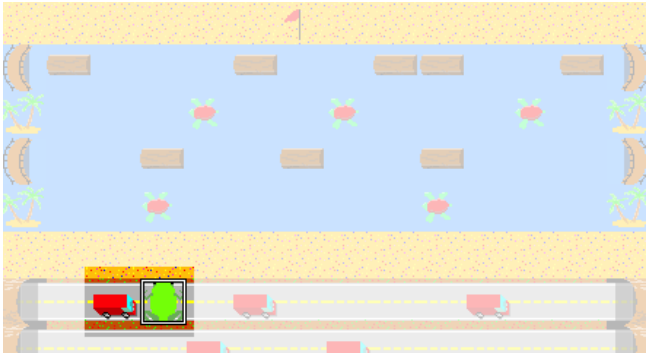
Conversational Programming as presented here is integrated into the AgentSheets game and science simulation end-user programming tool [6] used in schools world wide. Visual AgenTalk is the drag and drop, rule-based programming language of AgentSheets—this language has a long history in educational applications that goes back to 1994.

Novices, such as middle school students building their first game with no programming background, as well as more advanced programmers, such as computer science undergraduate students, often have difficulties when trying to fully understand complex rules. For instance, confusion in understanding the significance of instruction sequences is surprisingly common and is not limited to beginning programmers [1]. Common questions include: why does this rule fire? Why does that rule NOT fire? Why is this condition or rule not even being tested? What is the order in which conditions and rules are tested? Why is the rule and condition order of fundamental importance?

Conversational Programming could be considered a *prebugging* tool [2] that provides answers to the questions listed above even before the program is completely written or executed. The Conversational Programming Agent (CPA) will proactively execute parts of the program as created by the programmer and annotate the program discretely in order to help the end-user recognize potential differences between the intended program and the actual program. A simple feedback approach based on subtle colors is employed to avoid issues of cognitive overload recognized by Hundshausen, with the Alvis system [7]. He suggested that cognitive overload might be a limiting factor that should be considered when designing programming feedback systems. The CPA focuses on the agent selected by the user in the game world and visualizes the outcome of running the existing program of the selected agent one step into the future. For instance, if the programmer had previously selected the only frog in the worksheet (Figure 2, left) then Conversational Programming annotations would suggest that the frog is about to be crushed by the truck.

III. Related Work

The asymmetrical conversation between programmers and programming environment has its roots in early programming approaches. Some of the first programming environments hardly included any kind of meaningful feedback, which turned the process of programming into a



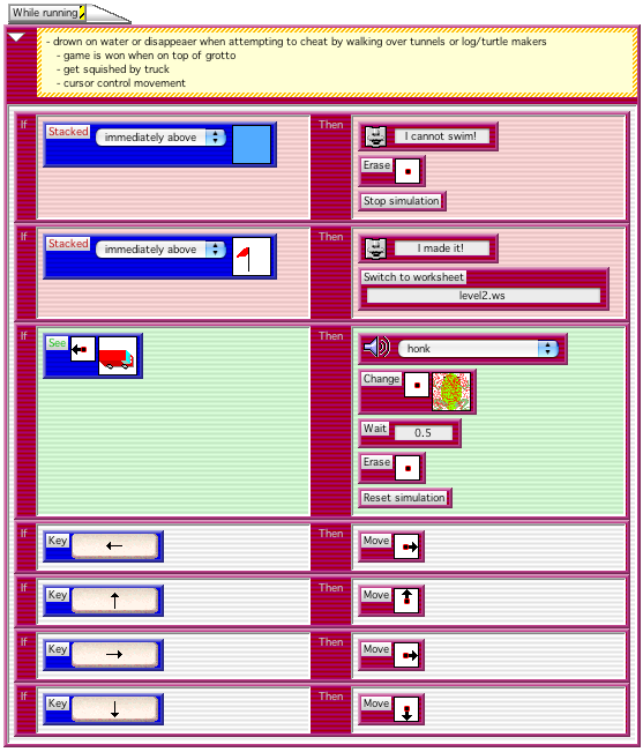


Figure 2. The truck will crush the frog selected in the worksheet (left). Rules 1 and 2 of the Frog behavior (right) are tested but contain at least one condition keeping them from firing. All conditions of rule 3 are true. Rules annotations (background): green=would fire, red=would not fire, and gray=not tested; Conditions annotations (text label): green=is true, red=is false, black=not tested.

complete monologue. A programmer would have to enter a complete, self-contained program all at once, and would not get even syntactic feedback. Only when trying to run or compile the entire program would the programmer find that the program failed to work. In the best-case scenario, the programmer might get some cryptic error message from the compiler. The obvious problems with this programming approach were recognized early, and researchers tried to create programming environments that would provide more immediate and more meaningful feedback. By 1967 the Dialog system [8] employed a variety of input/output devices, including switches and oscilloscopes, to provide feedback. This system was many years ahead of its time, and provided almost instant feedback to the programmer after each character input in a way similar to the much more modern code auto-completion found later in Integrated Development Environments. Interestingly, the Dialog system was already conceptualized as a "conversational programming system." The notion of picturing the interaction between a programmer and a programming environment as a conversation was explored early on and has been revisited often over the years.

The Lisp programming language has long included mechanisms that let programmers test not only complete programs but also test program fragments. In contrast to programming schools that advocate top down approaches—starting with a complete plan working towards an implementation—the Lisp philosophy encourages the programmer to start programming experimentally before a complete plan has been devised. The ability to run incomplete programs [17] in Lisp provides an efficient way to explore programs. DiSessa [18] calls the degree to which one is able to run a specific piece of code pokeability.

A very different approach to changing the nature of the conversation between the programmer and the programming environment, but with similar results, can be found in the field of visual programming [9, 10]. Instead of typing in text-based instructions, many visual programming languages use mechanisms such as drag and drop to compose programs. Similar to code auto-completion approaches, these kinds of visual programming environments prevent syntactic programming mistakes such as missing semicolons or typos. Systems such as AgentSheets [11, 12] provide dynamic drag and drop feedback to indicate compatibility/incompatibility between programming language building blocks as the user is trying to drag them onto targets. Other approaches use puzzle piece shaped programming language building blocks to convey compatibility. Some of these approaches go back as far as 1986 [13]. More recent systems aimed at end-users such as Scratch [14], Alice [15] and Squeak/eToys [16] employ similar approaches. AgentSheets and some of these related systems include the characteristic of pokability. However, we found the Conversational Programming approach to be significantly more effective because of: a) its *proactive nature*—programmers do not need to initiate the test of a condition, instead, the programming environment just shows if the condition is true/false all the time; and b) the *high degree of parallelism*—all relevant code will be annotated in real time.

Live programming is an attempt to reduce the cause / effect gap of programming by more tightly connecting a program with its environment. A program, in general, is not all that useful unless it is connected to some kind of environment. Flogo [19] is a programming language that annotates a running programming representation in various ways to indicate the state of the environment. For instance, the value of variables is presented in the program representation. Boolean expressions indicate if they are true or false when they execute. Live programming with SuperGlue [20] goes one step further by creating environment objects as the direct result of specifying code. For instance, a programmer defining a Pac-Man class and specifying its shape as a yellow disk would automatically get a yellow disk on the screen representing the Pac-Man.

Natural programming [4] explores a completely different way of providing semantic support. Natural programming is about creating programming languages that are closer to the way people think about tasks. Myers et al. have documented significant benefits for tasks such as debugging. However, in contrast to Conversational Programming natural programming does not include active mechanisms such as the Conversational Programming Agent to reduce discrepancies between the intended program and the actual program.

IV. Assessment

Conversational Programming has been integrated into AgentSheets 3, which was released in 2010. Our experience with the various debugging mechanisms integrated into AgentSheets over the last 15 years suggested focusing primarily on motivational and not usability concerns. In other words, it was not so much whether or not users *could* use a certain debugging mechanism, as whether or not they *would* actually employ the mechanism in practice. Observations were conducted in some of the Scalable Game Design project [21] test sites (mostly Colorado, Wyoming and South Dakota). With over 4000 mostly middle-school students participating in inner city, remote rural and Native American communities, the Scalable Game Design project has provided insight into how to bring the practice of debugging into highly diverse educational environments. The main finding to date is that the role of teachers and teacher training is even more important than initially assumed. Teachers need to be explicitly informed that debugging approaches in general, and Conversational Programming in particular, are not just additional features but are fundamental computational thinking [22] skills that will help with programming. The Scalable Game Design summer institutes have therefore gradually increased the

percentage of time spent on debugging practice, including sessions on Conversational Programming. We have found this teacher training to be well spent, and have seen the number of teachers and students using Conversational Programming as a debugging aid grow quickly in schools.

At the University level, we received feedback on Conversational Programming from Computer Science students through questionnaires completed after creating four different games using AgentSheets. We were interested in finding if a system like Conversational Programming, which was originally aimed at beginning programmers, would be appreciated by much more experienced programmers—or would it simply get in their way. The undergraduate students indicated that they kept Conversational Programming turned on (90%, n=10) and found that Conversational Programming was "very useful for debugging" (80% strongly agree, n=10). Some even expressed the wish to add Conversational Programming to programming languages such as C and Java.

V. Conclusions

Conversational Programming is a new model of interaction between programmers and programming environments. Unlike most drag and drop program composition models Conversational Programming is not limited to syntactic feedback, but also provides rich semantic feedback about programs by constantly executing and annotating them. While evaluation is still at an informal stage, the initial results are very encouraging and indicate that Conversational Programming could profoundly change how we conceptualize programming and debugging.

VI. Acknowledgements

This material is based in part upon work supported by the National Science Foundation under Grants DLR-0833612 and IIP-0848962. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

VII. References

1. Pea, R. D. Chameleon in the Classroom: Developing Roles for Computers, Logo Programming and Problem Solving. In Proceedings of the American Educational Research Association Symposium (Montreal, Canada, April 1983) (Montreal, Canada, 1983)
2. Lieberman, H. Your Wish Is My Command: Programming by Example. Morgan Kaufmann Publishers, San Francisco, CA, 2001.
3. Cypher, A. Watch What I Do: Programming by Demonstration. MIT Press, Cambridge, MA, 1993.
4. Myers, B. A., Pane, J. F. and Ko, A. Natural programming languages and environments. Communications of the ACM, 47, 9 2004), 47-52.
5. Telles, M. and Hsieh, Y. The Science of Debugging. Coriolis Group Books, Scottsdale AZ, USA, Scottsdale, 2001.
6. Repenning, A. and Ambach, J. Tactile Programming: A Unified Manipulation Paradigm Supporting Program Comprehension, Composition and Sharing. Computer Society, City, 1996.
7. Hundhausen, C. D., Farley, S. and Lee Brown, J. Can Direct Manipulation Lower the Barriers to Programming and Promote Positive Transfer to Textual Programming? An Experimental Study. IEEE Computer Society, Washington, DC, USA, City, 2006.
8. Cameron, S. H., Ewing, D. and Liveright, M. DIALOG: a conversational programming system with a graphical orientation. Communications of the ACM, 10, 6 1967), 349-357.
9. Burnett, M. Visual Programming. John Wiley & Sons Inc., 1999.
10. Shu, N. Visual Programming. Van Nostrand Reinhold Company, New York, 1988.
11. Repenning, A., Ioannidou, A. and Zola, J. AgentSheets: End-User Programmable Simulations. Journal of Artificial Societies and Social Simulation, 3, 3 2000).
12. Repenning, A. and Ioannidou, A. What Makes End-User Development Tick? 13 Design Guidelines. Kluwer Academic Publishers, City, 2006.
13. Glinert, E. P. Towards "Second Generation" Interactive, Graphical Programming Environments. Computer Society Press, City, 1986.
14. Resnick, M., Maloney, J., Monroy-Hernández, A.*, et al.* Scratch: Programming for All. Communincation of the ACM, 52, 11 2009), 60-67.
15. Conway, M., Audia, S., Burnette, T.*, et al.* Alice: Lessons Learned from Building a 3D System For Novices. City, 2000.
16. Freudenberg, B., Ohshima, Y. and Wallace, S. Etoys for One Laptop Per Child. IEEE Computer Society, City, 2009.
17. Teitelman, W. History of Interlisp. City, 2008.
18. diSessa, A. A. Twenty reasons why your should use Boxer (instead of Logo). City, 1997.
19. Hancock, C. M. Real-time programming and the big ideas of computational literacy. Dissertation, Massachusetts Institute of Technology, 2003.
20. McDirmid, S. Living it up with a live programming language. ACM, City, 2007.
21. Repenning, A., Webb, D. and Ioannidou, A. Scalable Game Design and the Development of a Checklist for Getting Computational Thinking into Public Schools. ACM Press, City, 2010.
22. Wing, J. M. Computational Thinking. Communications of the ACM, 49, 3 2006), 33-35.