Conversational Programming

Exploring Interactive Program Analysis

Alexander Repenning[1,2]
AgentSheets Inc[1], Boulder, Colorado,
University of Colorado[2], Boulder, Colorado
ralex@cs.colorado.edu

Abstract

Our powerful computers help very little in debugging the program we have so we can change it into the program we want. We introduce Conversational Programming as a way to harness our computing power to inspect program meaning through a combination of partial program execution and semantic program annotation. A programmer in our approach interactively selects highly autonomous "agents" in a program world as conversation topics and then changes the world to explore the potential behaviors of a selected agent in different scenarios. In this way, the programmer proactively knows how their code affects program execution as they explore various contexts. This paper describes conversational programming through design principles and use cases.

Categories and Subject Descriptors D.1.5 [Object-oriented Programming]; D.1.5 [Visual Programming]; D.3.3 [**Programming Languages**]; D2.5 [Testing and Debugging]: testing tools; D.2.6 [Programming Environments]: Interactive Environments.

Keywords: *Game design, computational thinking, computational science, debugging, end-user programming, visual programming.*

1. Introduction

Although computers have become incredibly powerful, debugging programs is still an arduous task. Imagine that a programmer is working on a game or simulation based on many objects, but the program is not behaving correctly and requires debugging. Pea [27] conceptualizes the process of debugging as "systematic efforts to eliminate discrepancies between the intended outcomes of a program [the program we want] and those brought through the current version of the program [the program we have]." In order to test it, our programmer is playing her game while seeking to find a situation where what she expected to see does not happen. What went wrong? She starts looking at the code. How does her powerful multi Gigahertz, parallel-processing computer help her at this moment? Sadly, in spite of this power, her programming environment is providing little if any help. Should it not be possible to employ that enormous computational power to analyze the situation the game is in, and to provide semantic feedback on *what* the program is doing and *why* it is doing it?

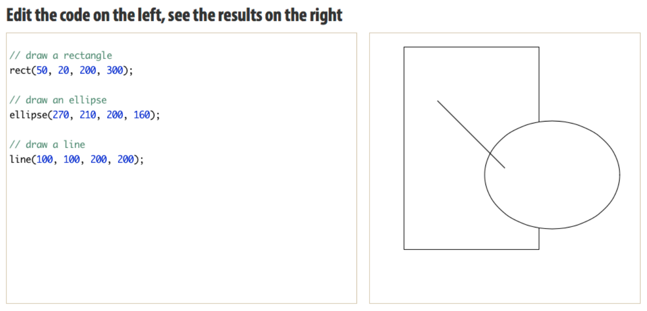
This is the goal of conversational programming: use the power of the computer to provide immediate semantic feedback to programmers. We believe that this is of particular relevance to non-expert programmers who generally have limited understanding of typically complex debugging tools.

Our motivation for this research comes from several large computer science education projects where we dealt with novice programmers. These novices included elementary school students using our AgentSheets [28, 37, 40] and AgentCubes [13, 14, 34, 38, 42] programming environments. Our project goal was to teach these students computational thinking [21, 50] by having them create games and simulations. In the context of the Scalable Game Design project [33], we have worked with over 10,000 students and found debugging to be one of the largest challenges for computer science students and teachers alike. This challenge arises, not because existing environments do not have usable debugging tools, but because these tools need to become much more proactive to become truly useful.

An ideal tool would support debugging by visualizing discrepancies between the "the program we want" and the "program we have." This ideal is not possible because "the program we want" only exists in the mind of the programmer and is not accessible to the computer. The best the computer can do is to explicitly present the semantics of the "program we have" to programmers and prompt them to experience potential discrepancies. Programming approaches that help avoid making mistakes in the first place are a step in the right direction.

Visual programming [2, 47] approaches and, more generally, approaches such as structured editing can make programming more accessible to novice programmers by reducing some of their syntactic programming struggles with problems like missing semicolons. With AgentSheets, we pioneered a number of visual programming approaches that included programming by example [29] and educational drag and drop programming [35]. Our goal was to make programming accessible to young children. However, the value of such approaches is limited [19]. Just as spell checkers do not automatically turn people into best-selling authors, visual programming does not guarantee that programs will make sense or even work at all.

Live programming is concerned with the semantic level of programming. McDirmid defines [23] "Live programming eliminates disruptive debugging sessions by allowing us to edit and execute code concurrently." Live programming is useful for a number of applications including user interface programming [1]. If a programmer makes a mistake, how long will it take for the resulting problem to manifest itself? A popular example of a simple live programming is the programming environment featured by the Khan Academy (Figure 1). A programmer can, for instance, change the position and size of the rectangle by editing the parameter values of the `rect` function by typing in new values or using a value slider. As the value changes the drawing updates instantly—making the connection between code and result crystal clear. According to experimental psychologist Michotte, if reporting can be done in just a few milliseconds [24] this may further help programmers perceive the connection between cause and effect. Michotte calls this the "perception of casualty" and suggests that this kind of observation is not a cognitive process with clear and active thought processes, but instead is experienced as a direct causal connection.


Edit the code on the left, see the results on the right

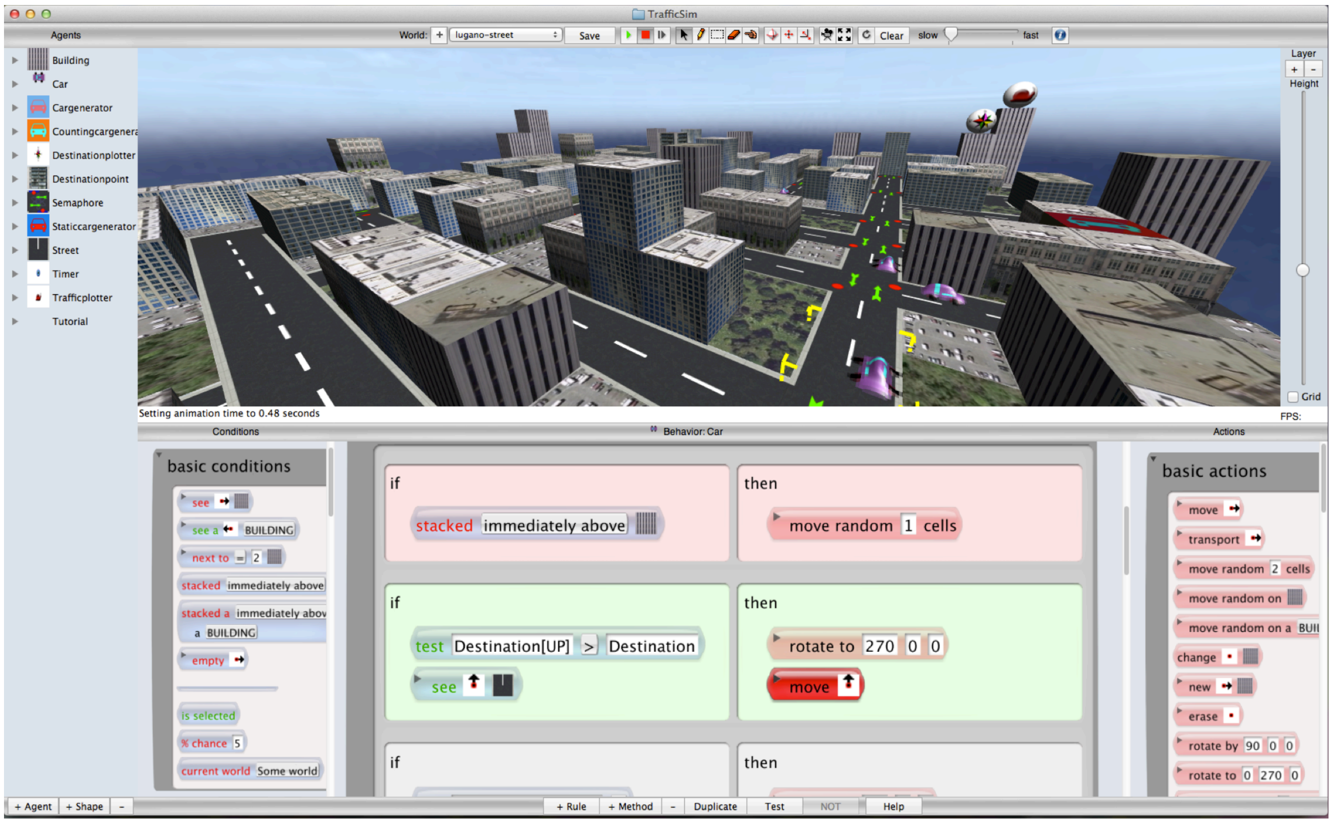
```
// draw a rectangle
rect(50, 20, 200, 300);

// draw an ellipse
ellipse(270, 210, 200, 160);

// draw a line
line(100, 100, 200, 200);
```

Figure 1. A Live Programming environment at Khan Academy

The basic model of live programming is quickly challenged when programs become more complex [23], do not converge, do not terminate, include non-determinism or depend on user interaction. An AgentCubes simulation (Figure 2) consists of a potentially large number of agents, and a world containing instances of these agents. AgentCubes is a live programming environment in the sense that when the programmer is running the simulation, any change to the program is instantly reflected in the program execution—just as with the program in Figure 1. However, there are at least two cases in which this kind of "liveness" [20]may be either unwanted by the programmer or impossible to achieve because of computational constraints.

- ***Unwanted liveness***: Executing the entire program to show the consequences of a change may be unwanted by programmers if they are engaging in a series of changes that may produce inconsistent in-between states of the program that could create unwanted side effects.
- ***Intractable liveness:*** A program could include loops that may be unbounded or may result in significant delays. This would decrease the usefulness of live programming [23]. Also, one might create non-determinisms that do not lead to a well-defined single converging future, but instead imply multiple different futures. Non-determinism makes long term program forecasting difficult.

We propose *conversational programing* as an extension of the live programming framework. Conversational programming applies live programming concepts to applications that are based on sets of highly autonomous non-deterministic objects such as agents. A conversation includes the notion of a conversation *topic* and an



Setting animation time to 0.48 seconds

```
if                                          then
  stacked immediately above                   move random 1 cells
if                                          then
  test Destination[UP] > Destination          rotate to 270 0 0
  see                                         move
if                                          then
```

Figure 2. Selecting an agent in the AgentCubes world will bring up its behavior. Conversational programming runs the behavior one step into the future and annotates it to forecast what the agent will be doing. The left panel is the collection of agent classes. At the top is the world. At the bottom is the behavior editor with the condition palette on the left and the action palette to the right.

interactive, spontaneous communication between the programming environment and the programmer.

The topic of the conversation is an agent selected by the programmer in the world. In Figure 2, the programmer selected a car in a city traffic simulation. The selected car can be recognized through its yellow selection handles. Selecting the car brings up the behavior, i.e., the code, of the car and semantically annotates the behavior—forecasting to the programmer what rule of the behavior would run. This selection is essential, because the entire program analysis is done only for the specific selected agent instance. The programmer could have selected other cars of the same class with potentially completely different code annotations based on where the car is in the world and what its attribute values are. At the left of the behavior in Figure 2 is the Visual AgentTalk [37] conditions palette. This palette also gets updated based on the agent selected.

The interactive and spontaneous communication component of the conversation is based on the programmer's ability to not only change selections but also to change the world in order to explore the behavior of agents. By dragging a car around the programmer can move it to different parts of the city to explore different scenarios. In this case the programmer does not actually change the code, but interacts with the world by changing it in order to get information back about the behavior of an agent. For instance, the programmer may drag a car to a different traffic light and discover that that car would turn in the wrong direction.

Conversational programming contributes to the framework of live programming by adding the idea of object selection to direct the focus of live programing. This kind of "directness" [20] is not only useful to the programmer but makes live programming more applicable for agent-based applications that might include non determinism and user interaction. With this approach, only the conditions of a single agent instance need to be computed instead computing the entire program. Live programming is typically about "programming with feedback about program execution", while conversational programming is speculative in the sense of "programing with feedback about how the program could execute."

A previous paper [32] presented an early example of conversational programming, but did not provide a full conceptual framework. Section 2 of this paper describes the conversational programming architecture and includes design principles. Section 3 explains how conversational annotation works through use cases. Section 4 describes related work.

2. Conversational Programming Architecture

The conversational programming architecture presented here should not be understood as a universal architecture but as an early proof of concept approach to investigating the usefulness of conversational programming for end-user programming [5, 11, 16]. Adding conversational programming to AgentSheets and AgentCubes turned debugging from a user initiated reactive activity to a system initiated proactive one. AgentSheets programs are rule based (e.g., Figure 10) and include the notion of conditions and actions. A large percentage of student problems were based on confusion with the relevance of rule order and condition order and whether individual conditions were true or false. These kinds of semantic issues could have been easily explored with the built-in "test" button of AgentSheets, which allows any condition, action and rule to be tested individually. However, our experience is that the large majority of students failed to use the "test" button as a debugging tool. Even after explicit encouragement, students typically fell back into a mode where they would just guess which rule was the problem and then try to address the problem by tweaking that rule. Part of this difficulty may be that students perceive condition selection to be a lot of work. In order to find a problem, it may be necessary to select quite a few conditions to localize the problem source. The proactive nature of conversational programming overcomes this problem by evaluating all the relevant conditions of the selected agent in parallel, that is, without the need to individually select conditions and to press the test button for each one. In essence, conversational programing is doing a lot of the programmer's busy work.

The intention of adding conversational programming to AgentSheets was to implement a more proactive debugging approach that would help overcome the hesitancy of end user programmers to employ debugging tools. Conversational programming in AgentSheets essentially tries to run the program one step into future, but only to the point where it can show the programmer what would happen without actually making it happen. Conversational programming semantically annotates rules to show which rules will fail and which rules will fire. It also annotates all the conditions that had to be tested to make this determination.

Our design principles to integrate conversational programming into AgentSheets and AgentCubes were:

- ***Do not cause side effects***: While conditions need to be checked, actions should not be executed as they would have side effects. Most, but not all, conditions of the Visual AgenTalk language are side-effect free. The timer condition, for instance has a side effect that needs to be addressed.
- ***Be responsive***: When a user changes the program or a state relevant to the program, for instance by editing the world, the annotation should change nearly instantly. In AgentCubes, conditions that are flipping from false to true or vice versa even get animated to help with the perception of causality [24].
- ***Be CPU conservative***: This kind of program analysis is fully tractable. That is, there is no concern that the annotation will require some unbounded amount of time. Even so, programs can have a very large number of rules. Some conditions such as the WWWread condition in AgentSheets actually read and parse web pages on remote servers. Testing this kind of condition can take a long time that may only be bounded by networking time outs. Generally, the analysis runs in a separate thread to make use of multi core hardware and to minimize the impact on runtime behavior.
- ***Be non intrusive***: By conversation we do not mean an intrusive form of feedback. The user should never have to wait, and feedback should be subtle. Feedback should not be like a blinking Christmas tree. We are experimenting with subtle color annotations.
- ***Be context sensitive***: For instance, in an agent-based simulation the annotation should be about the behavior of the agent currently selected in the world and its program.

Conversational Programming: Conversational programming is a way to harness computing power to inspect program meaning through a combination of partial program execution and semantic program annotation. A programmer in our approach interactively selects highly autonomous "agents" in a program world as conversation topics and then changes the world to explore the potential behaviors of a selected agent. In this way, the programmer proactively knows how their code will affect program execution in the various contexts that they explore.

Figure 3 shows the conversational programming architecture. To make this kind of feedback possible the CPA will need to be able to execute the program and to have access to the situation—a combination of data as well as selection. The notion of conversation emerges from an interaction taking place between the programmer and the Conversational Programming Agent (CPA) through the program. The programmer may edit the program or edit the situation. The CPA, in turn, will execute the program, or selective parts of the program, in the context of the situation. The results of this execution will be represented through annotations in the program.

Conversational programming communicates what would happen if a program was executed. To achieve this, the CPA needs to be restricted to running code without causing side effects. That is, the CPA will run code that reads the program state, but is not allowed to run code that would change the state. However, the programmer does have an option to execute actions if needed. The CPA is autonomous. It will run code even if the main program, i.e., the game or simulation, is not currently running. The CPA runs in its own thread, which, on multi-core machines, causes minimal overhead.

A simple example is the execution of conditions, which return a value of either true or false. The results of the execution are used to annotate these conditions—red for conditions being false and green for conditions being true. The large size of the semantic feedback arrow pointing back at the programmer reflects the rich nature of the conversational feedback that is provided. Even a relatively small conversation starter, such as the programmer applying a minor program edit, may create a large amount of semantic feedback. For instance, just slightly moving the frog in a Frogger like game (e.g., Figures 4-6) may result in many annotation changes in the program.

The semantic support of conversational programming is not achieved by the computer comprehending the meaning of the program. Instead, semantic support is achieved by establishing a tight loop from user input. This is done through program annotation based on the state of the running program that is directed back to the user in a way that makes the semantic consequences of the program immediately visible.

The components of the conversational programming environment architecture are listed below. Where necessary, AgentSheets [39, 41] is used as an illustration to make examples more concrete. AgentSheets is an agent-based simulation and game-authoring tool. The principles of conversational programming hold true for any kind of object-oriented or agent-based computational system. Agents are autonomous objects that can be implemented with any object-oriented system featuring some kind threading mechanism. The main components of the conversational programming environment architecture are:

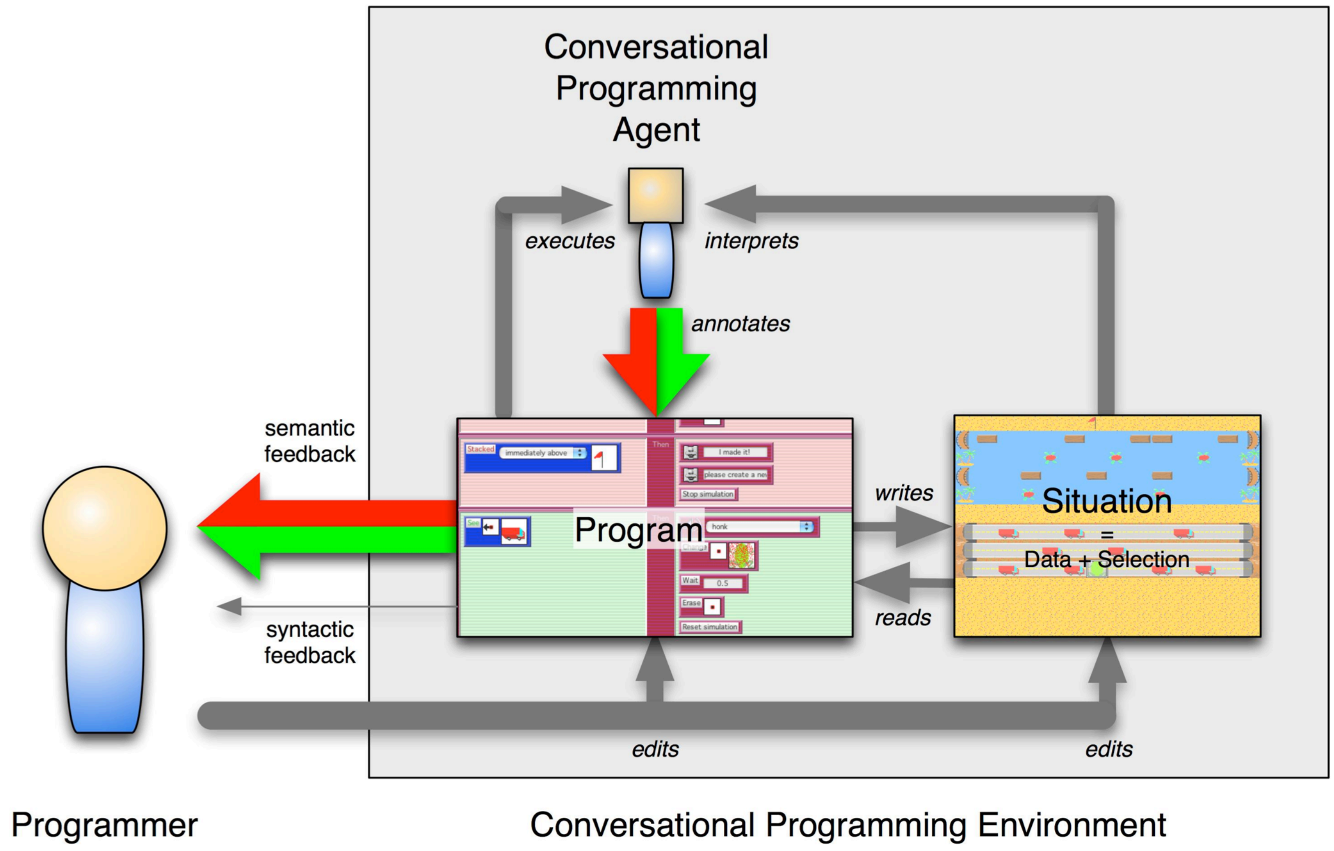


Figure 3. Conversational Programming. A Conversational Programming Agent (CPA) executes the program and provides rich, semantic level feedback to the programmer relevant to objects of interest to the programmer.

- ***Situation = Data + Selection***. The situation describes the combination of *data* and *selection*. Many end user programming environments include the notion of a situation capturing a collection of objects and some selection. Different environments use different terms such as stage, worksheet, or world. Data, similar to the notion of data in a spreadsheet, describes the collection of all agents in worksheets. Most of the data manifests themselves visibly to the user. Such data might include the position, size or shape of an agent. Other data such as agent attributes might not have a visible manifestation and would only become accessible to the user through tools such as inspectors. Selection designates a single agent to be the object of semantic investigation. A user selects an agent by clicking on it. The combination of selection and data is called the *situation* specifying the topic of the conversation. For instance, if the user selects the frog in a Frogger game then the conversation will be focused on that selected frog as it currently exists in the context of the game.
- ***Program***. The program expresses the function of a project. This function is a collection of all the behaviors expressed as methods of agent classes. In AgentSheets, programs are based on Visual AgenTalk [39, 41] and include notions of methods, rules, conditions, triggers and actions. The CPA will execute programming language building blocks in the context of the current situation and annotate these blocks. Additionally, the CPA annotates program fragments containing these blocks. Consider a rule with conditions c_1 & c_2 & … c_n and some actions. If any of these conditions are false then the CPA will annotate not only that condition but also the entire rule because the CPA can conclude that the rule would not fire.
- ***Conversational Programming Agent (CPA)***. The CPA communicates with the programmer by annotating programming language building blocks. It does this by executing them in the context of the current situation. The programmer communicates with the CPA by changing the situation, i.e., editing data or changing selection, by changing the program or by changing parameters of programming language building blocks.

3. Conversational Programming in Action

Examples will help to illustrate the function of conversational programming. Unfortunately, the intrinsically static nature of the paper used to represent these examples poorly conveys the intrinsically dynamic nature of conversational programming.

1.1 Latent Programs

Many end-user programming environments including Scratch [43], Squeak/EToys [7], Alice [4], and AgentSheets include programming palettes (libraries) of building blocks (see also in Figure 2, the Conditions and Actions palette). The main purpose of these palettes is to allow end users to browse building blocks, explore them and employ them by dragging and dropping them into operational programs. These palettes can be considered *latent programs* in the sense that they do contain valid fragments of programs that could be executed. For instance, building blocks representing conditions could be tested to see if the conditions are true or false. The CPA (Conversational Programming Agent) can annotate latent programs. This annotation may help with the process of browsing to find programming building blocks of particular relevance for a specific situation.

In the first example a Frogger-like game is built with AgentSheets. The programmer has created a number of agents—including a frog, trucks, roads, and ground. The programmer has created a situation representing the game world (background of Figures 4-6) and has made the frog the current selection. Now the programmer is starting to program the frog using the Visual AgenTalk [36] visual programming language built into AgentSheets. Figures 4, 5, and 6 show the reactive change in annotation of the latent programming language building blocks contained in the AgentSheets conditions palette as the programmer explores a number of scenarios by dragging the frog around in the worksheet. In the foreground is the Conditions palette. The Conditions palette contains many more conditions, all of which are annotated in the context of the selected agent (see Figure 2). Parameters of these conditions are universal, that is, they work for all agent classes. Directness [20] is achieved by having the conditions updated the moment the programmer selects a different agent instance in the world. A programmer noticing that a certain condition has turned true in the palette may find this information useful when employing this condition in the behavior of the agent.

An earlier version of AgentSheets and other end-user programming tools such as Scratch [43] and Alice 3D [4] include a Test button which allows programmers to select an agent/object and a programming language building block such as a condition to be tested. This lets a programmer determine if a condition is true or false. However, conversational programming substantially improves on two important challenges to this approach. First, the programmer does not need to ask for this kind of feedback. The system *reacts immediately* to a change in situation. Second, the programmer does not have to select a specific language building block for evaluation. Instead, *all the language blocks relevant to the situation will be annotated* automatically by the CPA.

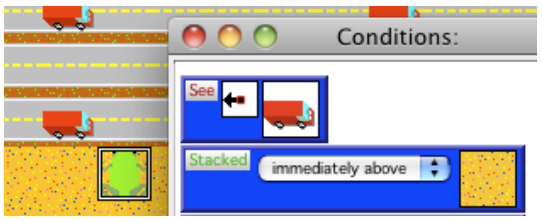


Figure 4. Frog is about to cross the street. Stacked (immediately above, ground) is true; See (left, truck) is false

At the very least this is an improvement in efficiency. A programmer could sequentially select all the conditions in the condition palette and repeat to test. However, this could take a long time (e.g., AgentSheets includes about 30 different conditions). The CPA, conceptually speaking, will annotate all the building blocks in parallel. This approach is not only much faster but it also increases the potential of *serendipitous discovery*. After all, a programmer may not even be aware that a certain programming language building block exists, or that it features relevant semantics. Based on timing alone the ability to quickly change the situation and to almost immediately perceive semantic consequences can result in the perception of causality [25] in a way that was not possible with previous mechanisms.

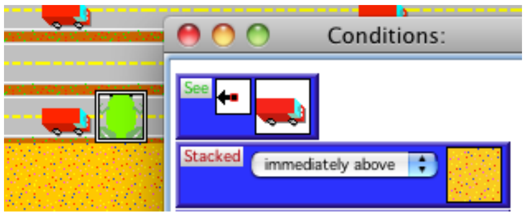


Figure 5. Frog is on street next to truck. Stacked (immediately above, ground) is false; See (left, truck) is true

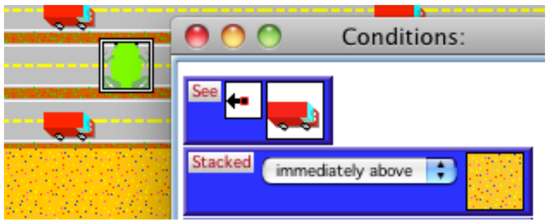


Figure 6. Frog is on street without a truck heading towards it. Stacked (immediately above, ground) is false; See (left, truck) is false

One may wonder what the value of this feedback to the user really is. After all, if users understand the basic semantics of a particular condition then they should be able to determine its truth-value. However, it frequently turns out that users create similar, hard to distinguish shapes. Another frequent problem is that users may include really small or even invisible agent shapes and pile them up in stacks intentionally or accidentally. No matter how a user got to the point of believing that a certain condition should be true or false, it is often incredibly difficult to find bugs because of precisely these kinds of assumptions. The resulting blind spots are extremely hard to overcome because they have been ruled out in the search. Even at this low level of annotating individual conditions, conversational programming can be surprisingly useful because it helps to overcome this blind spot problem. Moreover, catching these low level problems as early as possible is important, because once such a condition is embedded in a large program it is even more difficult to track down.

1.2 Rules: Order of Execution

The consequence of rule order is unexpectedly difficult to understand for beginning, and sometimes even for experienced, programmers.

Conversational programming annotates rules red, green or white/neutral (Figure 7). Rules are tested top to bottom identically as `IF THEN ELSE IF…` statements in most programming languages such as Java. A red rule would be tested but cannot fire because there is at least one false condition. With a list of n rules there could be n red rules. A green rule would fire because all of its conditions are true. There can only be 1 or 0 green rules. White/neutral rules are not tested at all because they are preceded by a firing rule. There can be n-1 white/neutral rules.

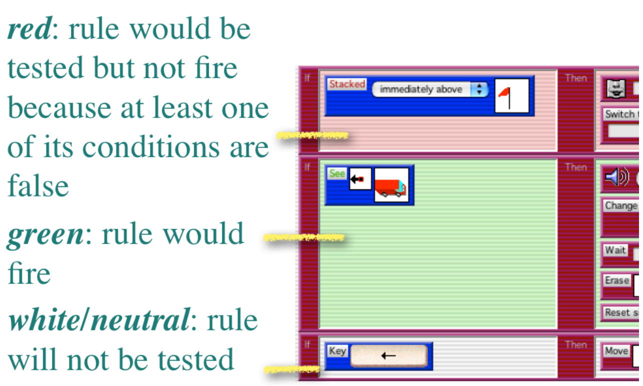

red: rule would be tested but not fire because at least one of its conditions are false
green: rule would fire
white/neutral: rule will not be tested
If
Stacked
immediately above
Then
Switch
See
Change
Wait
Erase
Reset
Key
Move

Figure 7. Red, Green, White/neutral rule annotation.

Conversational programming annotates conditions red, green or black/neutral similarly to the annotation of rules (Figure 8). Conditions are tested top to bottom, again just like in most programming languages such as Java. All conditions need to be true for a rule to be able to fire. A green condition is true. In a rule with n conditions up to n conditions can be green and would have be green for the rule to become green. A red condition is false. Only 1 condition per rule can be red because no other condition following that condition in the same rule would be tested. A back/neutral condition would not be tested at all. Up to n-1 conditions can be black/neutral.

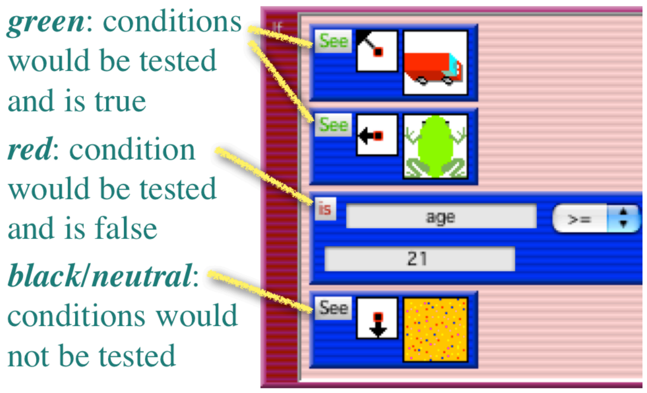

green: conditions would be tested and is true
red: condition would be tested and is false
black/neutral: conditions would not be tested
If
See
See
is
age
>=
21
See

Figure 8. Red, Green, Black/neutral condition annotations.

How do these annotations look in the context of a complete game? Lets assume the programmer is building a Frogger-like game (Figure 9). In the first scenario (Figure 10, top) the question is why does the frog get killed? The frog was selected and the rules were annotated accordingly. A number of rules are tested, including the drown rule (if the frog is stacked above water) and the reach the goal rule (if the frog is stacked above the goal), but all of them are false. Finally, the rule in which the frog is checking for a truck to its left is true.

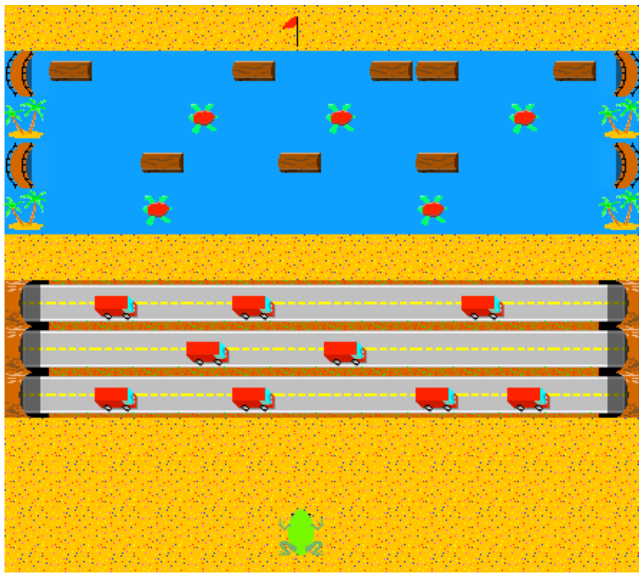


Figure 9. Simple Frogger-like game built in AgentSheets.

Programmers could execute the rule by double clicking it. If executed the frog would play a honk sound, change its appearance to be the bloody squished frog, wait for 0.5 seconds, erase itself, and reset the simulation. All this is expected and confirms the programmer's expectation. The cursor control rules following the collision rule are not tested. Had these rules preceded the collision rule then the player of the game would have had a chance to escape the approaching truck. In other words, the player could have cheated. The cursor control rules should be after the collision rule.

In our second scenario (Figure 11) there is a similar situation with the frog and the truck but the frog does not get killed. Why? Conversational programming suggests that the collision rule is not actually being tested. Instead, the preceding goal rule is true. This rule will make the game announce a win and switch to the second level. Only when looking at the situation very carefully does one see the goal flag nearly covered up by the frog. The difference between this situation and the one before is hard to see visually, but thanks to conversational programming it is clear what the frog is actually doing.

It might seem that these scenarios are quite simple, but in our experience these cases are not only extremely frequent but often lead to lengthy and frustrating debugging sessions. The problem is that the programmer, once convinced that a certain rule should fire, is really hard to dissuade from this theory. Even with help the false theory is often really hard to overcome. However, conversational programming makes it clear which rule will fire and often helps the programmer identify the difference between the program they want and the one they have.

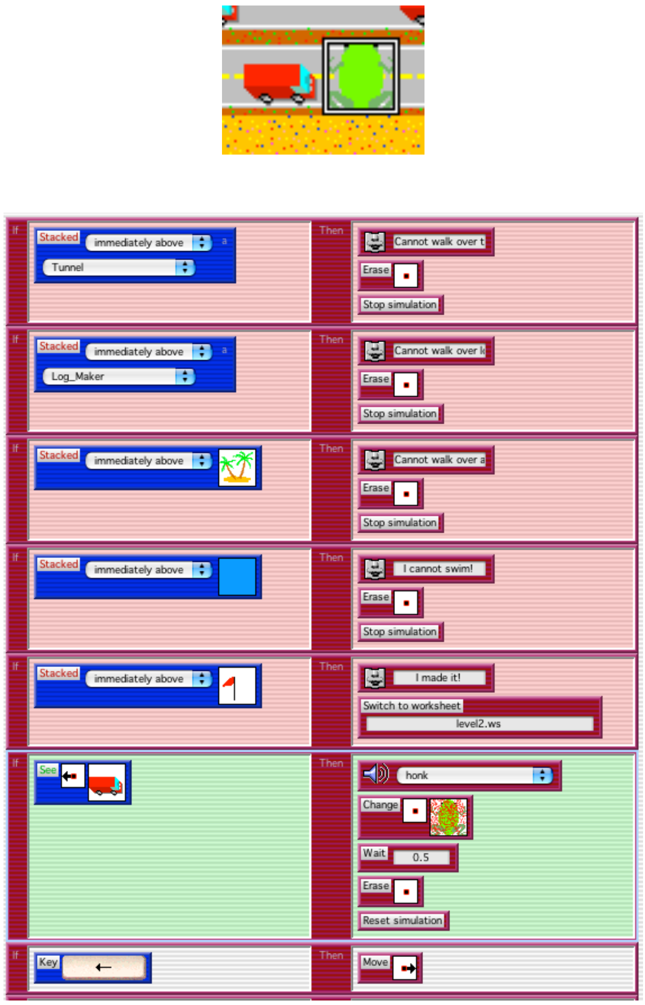
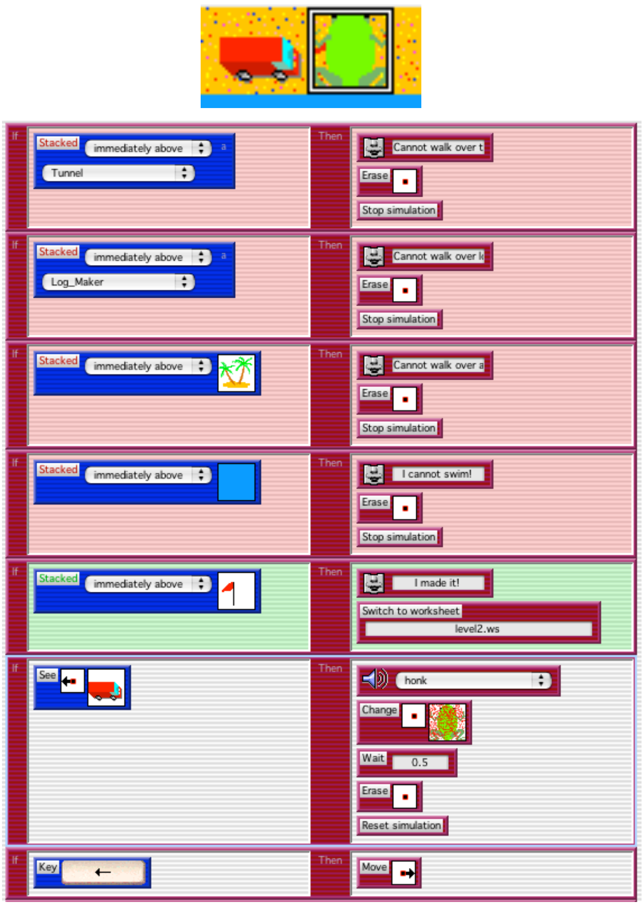


Figure 10. Why does the Frog get killed?

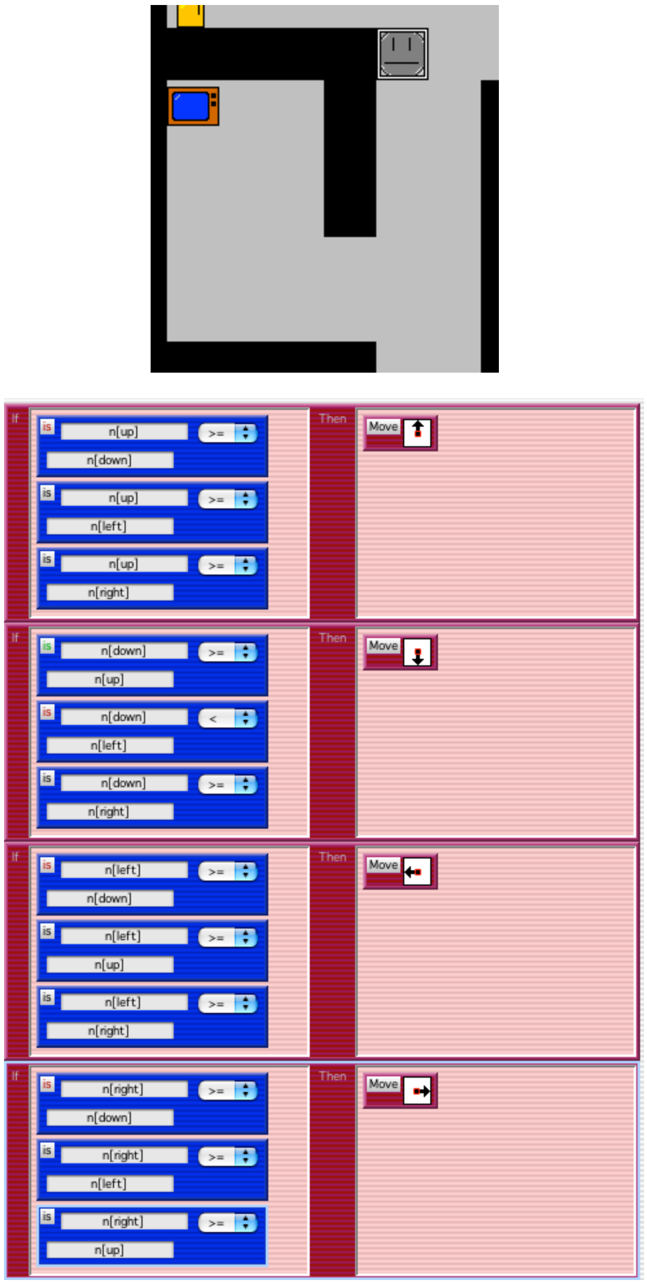
The scenario in Figure 12 is a bit more advanced. A middle school student has programmed some path finding AI based on collaborative diffusion [31]. A smiley face looking Mr. Sim agent is trying to find a path to entertainment represented as a TV in a room. In collaborative diffusion the TV would be at the peak of an entertainment surface mountain which the Mr. Sim character is supposed to hill climb. In Figure 12 Mr. Sim is supposed to move down but does not. Rule number two is the one expected to be running, but instead none of the rules fire. A closer look at all the conditions in the second rule indicates that condition number two is false (its label "is" is red). The programmer had selected the wrong comparator type. It should have been ">=" but was "<". The programmer fixes the bug by selecting the right comparator, the condition turns green, the following condition turns green as well, and then the entire rule turns green. The program is fixed and Mr. Sim is finally moving towards the TV.



```
If  Stacked  immediately above  a  Tunnel       Then  Cannot walk over t
                                                      Erase
                                                      Stop simulation
If  Stacked  immediately above  a  Log_Maker    Then  Cannot walk over l
                                                      Erase
                                                      Stop simulation
If  Stacked  immediately above                  Then  Cannot walk over a
                                                      Erase
                                                      Stop simulation
If  Stacked  immediately above                  Then  I cannot swim!
                                                      Erase
                                                      Stop simulation
If  Stacked  immediately above                  Then  I made it!
                                                      Switch to worksheet  level2.ws
If  See                                         Then  honk
                                                      Change
                                                      Wait  0.5
                                                      Erase
                                                      Reset simulation
If  Key  ←                                      Then  Move
```

Figure 11. Why does the Frog not get killed?

The next step of conversational programming is the dynamic annotation of non-deterministic programs. Imagine a ladybug to be programmed with a simple behavior of moving left or right randomly with equal probability. Figure 13 shows the two rules of the ladybug. The fifty percent chance condition, in the first rule, introduces non-determinism. The programmer arranged nine ladybugs as a single column into the world. Then the simulation ran a couple of simulation cycles. The annotation of the behavior has no static solution. For each single bug there are two possible futures. It will either move to the left or move to the right. The 50% condition could be true—in which case it would be green, the first rule would be green, and the second rule would be neutral. Or the fifty percent condition could be false. In this case the condition would be red; the first rule would be red, and the second rule would be green because it does not have any conditions. Notice the second rule does not need a fifty percent condition. If it did have such a condition the lady bug would only move to the left with a 0.5 x 0.5 = 25% chance, which would introduce a significant drift of the ladybugs to the right.



```
If  is  n[up]    >=  n[down]     Then  Move ↑
    is  n[up]    >=  n[left]
    is  n[up]    >=  n[right]
If  is  n[down]  >=  n[up]       Then  Move ↓
    is  n[down]  <   n[left]
    is  n[down]  >=  n[right]
If  is  n[left]  >=  n[down]     Then  Move ←
    is  n[left]  >=  n[up]
    is  n[left]  >=  n[right]
If  is  n[right] >=  n[down]     Then  Move →
    is  n[right] >=  n[left]
    is  n[right] >=  n[up]
```

Figure 12. The sim is not moving down.

AgentCubes employs a more sophisticated version of conversational programming and includes animated annotations. If a condition changes from true to false or false to true then this change will be animated by flipping the condition in the code to get the attention of the programmer. This addition enhances the effect of "perception of causality" [25] as suggested by Michotte.

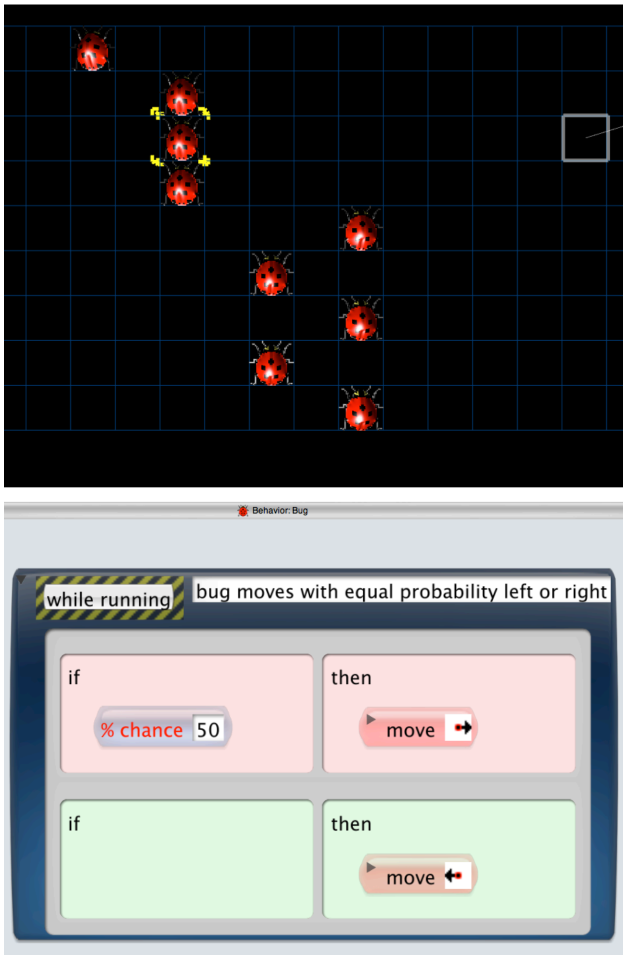


Figure 13. Programming ladybugs to move left or right randomly. The red/green annotation dynamically indicates that sometimes rule number one and sometimes rule number two will fire.

4. Related Work

Many early programming environments only included limited feedback. A programmer would enter a complete program and would not get syntactic feedback. Then, when trying to run or compile the program the programmer would see that the program does not work. In the best-case scenario, there might be some error message from the compiler. The main problem with this programming approach was recognized early. Researchers started to create programming environment systems that would provide some meaningful feedback. In 1967 the Dialog system [3] employed input/output devices like switches and oscilloscopes to provide almost instant feedback to the programmer after each character input. This system was well ahead of its time and operated in a way similar to the modern code auto-completion found later in Integrated Development Environments. Interestingly, the Dialog system was already conceptualized as a "*Conversational Programming system*." Over the years, the conceptualization of the interaction between a programmer and a programming environment as conversation has been revisited often.

Most of the explorations of conversational programming had the shared goal of making the programming process more reactive. Early examples include mechanisms such as the code auto-completion found in Lisp machines [45]. Later implementations include tools such as IntelliSense of Visual Studio. Similar environments include Eclipse, and Xcode. All these environments are responsive to text input and help the programmer by either popping up valid completion menus or a projecting constrained set of characters and symbols on a virtual keyboard [3].

A very different approach to changing the nature of the conversation between the programmer and the programming environment, but with similar results, comes from the field of visual programming [2, 47]. Instead of typing in text-based instructions, many visual programming languages are using mechanisms such as drag and drop to compose programs. Similar to code auto-completion approaches, these kinds of visual programming environment essentially prevent syntactic programming mistakes such as missing semicolons. Systems such as AgentSheets [39, 41] provide dynamic drag and drop feedback to indicate compatibility/incompatibility of some programming language building blocks. Other approaches experiment with puzzle piece shaped programming building blocks to convey compatibility. Some of these approaches go back to 1986 [8]. More recent systems aimed at end-users such a Scratch [43], Alice [4] and Squeak/eToys [7] employ similar approaches.

The Lisp community has explored ideas of bottom up programming for some time. In contrast to programming schools advocating top down approaches that start with a complete plan and work towards an implementation, the Lisp philosophy encourages the programmer to start programming before a complete plan has been devised. The request to run incomplete programs [48] is an especially efficient means of exploring programs. DiSessa [6] calls the degree to which one is able to run a specific piece of code "pokeability".

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. A sorting program is used to sort a collection of numbers. Flogo [9] is a programming language that annotates 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 [22] 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 yellow disk would automatically get a yellow disk object on the screen representing the pac-man object. The idea of instant feedback of semantic information (information about the value of cell having been tested) has also been applied to spreadsheets [44].

Various models have been defined to conceptualize interaction between human and computers. Norman talks about the gulf of evaluation as a metaphor to investigate challenges created by computational representations that can be directly perceived and interpreted by humans [26]. This is somewhat applicable to the kinds of representations programming environment employ to provide feedback to programmers in case of problems. Sneiderman goes further with his notion of direct manipulation [46] to address the entire loop of user input, system processing and the generation of meaningful and timely feedback.

There are a number of ideas to simplify debugging. Ko's Whyline [17] is a powerful debugging system that allows programmers to go back from program output symptoms to the code causing the problem. This is very different from techniques such as putting print statements into code or enabling breakpoints, because both of these approaches require the programmer to have a good sense of which code is causing the problem. Whyline, in contrast, allows programmers to find that code by backwards navigation from symptom to cause. One disadvantage of Whyline is that programmers need to quit the application to be debugged after the problem has occurred to use the separate Whyline tool to navigate back. This could be a workflow concern, as it does not allow programmers to fluently go back and forth between programming and debugging tools. However, this concern could well be overcome with future versions of the Whyline tool. The larger conceptual difference between Whyline, as well as similar back trace oriented debugging tools, and conversational programming is the time of use. Whyline is aimed at debugging looking backward in time from effect to cause. Conversational programming is more aligned with the notion of *prebugging* looking forward in time. Telles et all. [49] coined the notion of prebugging. They define the goal of prebugging to "reduce the odds of making mistakes, and when we do make mistakes, to have the infrastructure in place to detect, identify and eliminated these mistakes quickly and painlessly". In this sense the Whyline work and conversational programming could be considered highly complementary, but conversational programming with its focus on very simple but integrated debugging interfaces is aimed more at end user programmers.

Hundshausen, with the Alvis system [12], suggest that cognitive overload may be a limiting factor that should be considered when designing programming feedback systems. Potentially, cognitive overload may have to be considered a trade off for cause and effect immediacy. However, taking into account Michotte's [25] perception of causality argument the answer may not be to slow down reactions but instead to explore subtle kinds of feedback that allow users to experience cause and effect through real time visualizations.

A step in the direction of creating a more symmetrical communication including elements of semantics is programming by example [18]. For instance, the play-in/play-out approach establishes a strong interaction between users, graphical user interfaces and formal behavior specification [10]. Programming by example systems making these representations explicit to the user, such as the play-in/play-out approach, strongly overlap with the notion of conversational programming. These approaches hinge on user input including the selection and manipulation of objects. In contrast to conversational programming, however, most programming by example systems, including play-in/play-out, will automatically create formal behavior representations for the users. Conversational programming neither constructs nor changes the formal behavior presentation. In fact, conversational programming simply semantically annotates the programs created by the users.

A recent renaissance of learning to program sites includes a number of sites employing some form of responsive programming. For instance Khan Academy has a computer science section that teaches JavaScript through simple live programming. However, in contrast to conversational programming there is no means to select an object in a world or to interactively edit an object to explore the object behavior in different scenarios. An example JavaScript to create a snowman includes a number of "ellipse" function calls to draw the various ellipse shaped body parts of the snowman. The user can select constants, such as the horizontal and vertical size of the ellipse, and even use a slider to adjust the value of the constant interactively. Essentially every change to the program will result in re-evaluating it, including the drawing part. In other words, this kind of feedback illustrates program semantics. It is not clear how well this approach would scale, and deal the challenges such as non-determinism.

5. Assessment

Conversational programming has been integrated into AgentSheets 3 and AgentCubes. AgentSheets is an educational programming environment used by students to learn about computational thinking [50] by building games and computational science simulations. The audience ranges from middle school students building simple Frogger-like games to Computer Science graduate students building Sims-like games that include sophisticated Artificial Intelligence [30].

A formal evaluation of AgentCubes [15], which essentially is the 3D cousin of AgentSheets featuring identical programming, confirmed high degrees of end-user programming accessibility and support for general problem solving. However, at this point only semi-formal studies have been conducted to assess the specific contribution of conversational programming added to AgentSheets and AgentCubes. The general project evaluation is assessing motivational and programming skills levels but is not correlating them to specific tool affordances including conversational programming.

The main assessment question is not about usability, i.e., *can it be used*, but *will it be used*. A formal experiment could shed some light onto usability. For instance, one could try to compare the debugging performance of subjects using conversational programming with the debugging performance of a control group. With previous versions of AgentSheets only featuring the "test" button to test conditions and actions we already knew that most users could use that button when instructed to do so but, on average, they just did not use the button. Similarly, we feel that using conversational programming is not hard. The interface is minimal and requires, based on our experience with students and teachers, only a brief introduction. The instruments to explore the "will they use it" question have more to do with ethnography than with usability. Ethnographic studies include instruments such as classroom observation. Our early observations are still early but positive simply based on the fact that users appear to keep conversational programming turned on most of the time and that we have seen students and teachers use it successfully.

There have been many teachers and students not using conversational programming even in situations where they could benefit from it. We are blaming our teacher professional development approach which only minimally addresses debugging and pushes the introduction of conversational programming to the very end of the workshop. We believe this should be completely turned around. The idea of conversational programming and the introduction to rule-based programing in AgentSheets could go hand in hand to cover topics like rule order that are notoriously hard for teachers and students to understand. Having debugging and conversational programming at the end of the workshop also makes it sound as if it is one more topic to cover which teachers will push on the backburner and, as a result, only cover it when there is spare time.

We do have one study exploring the longer-term use of conversational programming. University students participating in a one-semester game design class using AgentSheets indicated that they kept conversational programming turned on and found that conversational programming was very useful. Each student created nine complete games. The largest shared concern was that conversational programming would only be activated if a behavior editor was associated with the selected agent in the worksheet. For instance, if a programmer is looking at the behavior editor of the frog in a Frogger-like game but then interacts with the game by clicking other agents then the frog behavior editor will no longer annotate its code because the frog is no longer selected. A related problem mentioned was that the conversational programming feedback was only available for one agent at a time. AgentSheets allows opening multiple behavior editors but only the behavior editor associated with the currently selected agent will annotate its code. A more sophisticated model would be necessary to deal with multiple selections including ways to correlate selection and code. 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 languages such as C and Java. While this is a good idea the transition of conversational programming from the rule-based Visual AgenTalk to more traditional languages such as C or Java is not trivial. For instance, Visual AgenTalk conditions can be assumed to be without side effect. This assumption does not generally hold in most programming languages.

6. Conclusions

Conversational programming is an extension to the live programming model supporting the application of live programing ideas to non-deterministic agent-based computing applications. Using the power of the computer conversational programming forecasts the future of the agent by testing the conditions of the selected agent. Conversational programming adds the notions of a conversation topic and an interactive communication. Programmers define the conversation topic by selecting an agent instance in a world. The interactive aspect of the conversation is based on programmers changing the world, for instance by moving agents around, to test how agents will behave in different scenarios. The version of conversational programming presented here is an early example of a relatively simple implementation built into the AgentSheets and AgentCubes end-user programming systems. In spite of its simplicity we have seen a number of highly encouraging use cases in which end users such as teachers and students were able to debug difficult programs with the help of conversational programming.

7. Acknowledgements

This material is based in part upon work supported by the National Science Foundation under Grant Numbers No. 0833612 and DMI-0712571. 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.

8. References

1. Burckhardt, S., Fahndrich, M., Halleux, P. d., *et al.* It's alive! continuous feedback in UI programming. In Proceedings of the Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation (Seattle, Washington, USA, 2013). ACM, 95-104.
2. Burnett, M. Visual Programming. John Wiley & Sons Inc., New York, 1999.
3. 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.
4. Conway, M., Audia, S., Burnette, T., *et al.* Alice: Lessons Learned from Building a 3D System For Novices. City, 2000.
5. Cook, R. Full Circle: In the beginning, everyone was a programmer. Now, with powerfull user languages, everyone is a programmer again. BYTE, 15, 8 1990), 211-214.
6. diSessa, A. A. Twenty reasons why your should use Boxer (instead of Logo). City, 1997.
7. Freudenberg, B., Ohshima, Y. and Wallace, S. Etoys for One Laptop Per Child. IEEE Computer Society, City, 2009.
8. Glinert, E. P. Towards "Second Generation" Interactive, Graphical Programming Environments. Computer Society Press, City, 1986.
9. Hancock, C. M. Real-time programming and the big ideas of computational literacy. Dissertation, Massachusetts Institute of Technology, 2003.
10. Harel, D. and Marelly, R. Specifying and executing behavioral requirements: the play-in/play-out approach. Software and Systems Modeling, 2, 2 2004), 82-107.
11. Harrison, W. From the Editor: The Dangers of End-User Programming. IEEE Software, 21, 4 (July 2004), 5-7.
12. 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.
13. Ioannidou, A., Repenning, A. and Webb, D. Using Scalable Game Design to Promote 3D Fluency: Assessing the AgentCubes Incremental 3D End-User Development Framework. In Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '08) (Herrsching am Ammersee, Germany, Sept. 15-19, 2008). IEEE Press, 47-54.
14. Ioannidou, A., Repenning, A. and Webb, D. AgentCubes: Incremental 3D End-User Development. Journal of Visual Language and Computing, 20, 4 2009), 236-251.
15. Ioannidou, A., Repenning, A. and Webb, D. C. AgentCubes: Incremental 3D end-user development. Journal of Visual Languages and Computing2009).
16. Jones, C. End-user programming. IEEE Computer, 28, 9 (September 1995), 68-70.
17. Ko, A. J. and Myers, B. A. Finding causes of program output with the Java Whyline. In Proceedings of the Proceedings of the SIGCHI Conference on Human Factors

in Computing Systems (Boston, MA, USA, 2009). ACM, 1569-1578.

18. Lieberman, H. Your Wish Is My Command: Programming by Example. Morgan Kaufmann Publishers, San Francisco, CA, 2001.
19. Lister, R. Computing Education Research: Programming, syntax and cognitive load. ACM Inroads, 2, 2 2011).
20. Maloney, J. H. and Smith, R. B. Directness and liveness in the morphic user interface construction environment. In Proceedings of the Proceedings of the 8th annual ACM symposium on User interface and software technology (Pittsburgh, Pennsylvania, USA, 1995). ACM, 21-28.
21. Marshall, K. Was that CT? Assessing Computational Thinking Patterns through Video-Based Prompts. In Proceedings of the 2011 Annual Meeting of the American Educational Research Association (AERA) (New Orleans, LA, April 8-12, 2011)
22. McDirmid, S. Living it up with a live programming language. ACM, City, 2007.
23. McDirmid, S. Usable Live Programming. In Proceedings of the SPLASH Onward! (Indianapolis, Indiana, October 2013, 2013). ACM SIGPLAN,
24. Michotte, A. The perception of causality. Methuen, Andover, MA, 1962.
25. Michotte, A. The Perception of Causality. Methuen & Co. Ltd., London, 1963.
26. Norman, D. A. The Design of Everyday Things. MIT Press, 1998.
27. Pea, R. LOGO Programming and Problem Solving. In Proceedings of the Paper presented at symposium of the Annual Meeting of the American Educational Research Association (AERA), "Chameleon in the Classroom: Developing Roles for Computers" (Montreal, Canada, April 1983., 1983)
28. Repenning, A. Agentsheets: A Tool for Building Domain-Oriented Dynamic, Visual Environments. Department of Computer Science, University of Colorado at Boulder, 1993.
29. Repenning, A. Bending the Rules: Steps toward Semantically Enriched Graphical Rewrite Rules. IEEE Computer Society, City, 1995.
30. Repenning, A. Collaborative Diffusion: Programming Antiobjects. City, 2006.
31. Repenning, A. Collaborative Diffusion: Programming Antiobjects. In Proceedings of the OOPSLA 2006, ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications (Portland, Oregon, 2006). ACM Press, 574-585.
32. Repenning, A. Making Programming more Conversational. In Proceedings of the 2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) (Pittsburgh, PA, USA, September 18–22, 2011). IEEE Computer Society, Los Alamitos, CA, 191-194.
33. Repenning, A. Programming Goes Back to School. Communications of the ACM, 55, 5 (May 2012), 38-40.
34. Repenning, A. Making Programming Accessible and Exciting. IEEE Computer, 18, 13 2013), 78-81.
35. Repenning, A. and Ambach, J. Tactile Programming: A Unified Manipulation Paradigm Supporting Program Comprehension, Composition and Sharing. In Proceedings of the 1996 IEEE Symposium of Visual Languages (Boulder, CO, 1996). Computer Society, 102-109.
36. Repenning, A. and Ambach, J. Visual AgenTalk: Anatomy of a Low Threshold, High Ceiling End User Programming Environment. Technical Report CU-CS-802-96, Department of Computer Science, University of Colorado, 1996.
37. Repenning, A. and Ioannidou, A. Behavior Processors: Layers between End-Users and Java Virtual Machines. In Proceedings of the Proceedings of the 1997 IEEE Symposium of Visual Languages (Capri, Italy, 1997). Computer Society, 402-409.
38. Repenning, A. and Ioannidou, A. AgentCubes: Raising the Ceiling of End-User Development in Education through Incremental 3D. In Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing 2006 (Brighton, United Kingdom, September 4-8, 2006). IEEE Press,
39. Repenning, A. and Ioannidou, A. What Makes End-User Development Tick? 13 Design Guidelines. Kluwer Academic Publishers, 2006.
40. Repenning, A., Ioannidou, A. and Zola, J. AgentSheets: End-User Programmable Simulation. Journal of Artificial Societies and Social Simulation, 3, 3 2000).
41. Repenning, A., Ioannidou, A. and Zola, J. AgentSheets: End-User Programmable Simulations. Journal of Artificial Societies and Social Simulation, http://www.soc.surrey.ac.uk/JASSS/3/3/forum/1.html, 3, 3 2000).
42. Repenning, A., Smith, C., Owen, B., *et al.* AgentCubes: Enabling 3D Creativity by Addressing Cognitive and Affective Programming Challenges. In Proceedings of the World Conference on Educational Media and Technology, EdMedia 2012 (Denver, Colorado, USA, June 26-29, 2012) 2762-2771.
43. Resnick, M., Maloney, J., Monroy-Hernández, A., *et al.* Scratch: programming for all. Communincation of the ACM, 52, 11 2009), 60-67.
44. Rothermel, K. J., Cook, C. R., Burnett, M. M., *et al.* WYSIWYT testing in the spreadsheet paradigm: an empirical evaluation. In Proceedings of the Proceedings of the 22nd international conference on Software engineering (Limerick, Ireland, 2000). ACM, 230-239.
45. Sandewall, E. Programming in an Interactive Environment: the ``Lisp" Experience. ACM Computing Surveys, 10, 1 1978).
46. Shneiderman, B. Direct Manipulation. A Step Beyond Programming Languages. IEEE Transactions on Computers, 16, 8 1983), 57–69.
47. Shu, N. Visual Programming. Van Nostrand Reinhold Company, New York, 1988.
48. Teitelman, W. History of Interlisp. City, 2008.
49. Telles, M. and Hsieh, Y. The Science of Debugging. Coriolis Group Books, Scottsdale AZ, USA, Scottsdale, 2001.
50. Wing, J. M. Computational Thinking. Communications of the ACM, 49, 3 2006), 33-35.